Hybrid Fault Tolerant Consensus in Wireless Embedded Systems

Von der
Carl-Friedrich-Gauß-Fakultät
der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigte Dissertation

von
Wenbo Xu
geboren am 31.12.1987
in Hubei, China

Eingereicht am:	09.03.2021
Disputation am:	24.06.2021
1. Referent:	Prof. Dr. Rüdiger Kapitza
2. Referent:	Prof. Dr. Franz J. Hauck

2021

Abstract

Consensus is a fundamental problem in distributed system. Nowadays cooperative autonomous systems gain increasing popularity, in which different participants can work in a coordinated way to achieve a common goal. Most of these systems demand for high fault-resilience, otherwise a single faulty node could render the whole system useless. This essentially calls for a Byzantine fault-tolerant consensus. However, typically only $\lfloor \frac{n-1}{3} \rfloor$ faulty nodes can be tolerated in a group of n nodes if the system is partially synchronous. This fault-tolerance rate is much lower than $\lfloor \frac{n-1}{2} \rfloor$ in crash fault-tolerance. Even worse, systems with only 3 nodes are too small to even tolerate a single Byzantine node. Since the Byzantine fault model where nodes can be arbitrarily faulty is too pessimistic, a more realistic hybrid fault model is considered in this thesis. In such a hybrid fault model, every node is equipped with a small trusted subsystem that can only be faulty by crashing, while the remaining part of the system can still be Byzantine. By exploiting the trusted subsystem, two consensus algorithms are proposed: TRUSTED BEN-OR is a binary consensus algorithm that can work in an asynchronous system, and RATCHETA is a multi-value consensus algorithm designed for partially synchronous systems. Both algorithms utilize the trusted monotonic counter(s) and improve the maximum tolerable faults to $\lfloor \frac{n-1}{2} \rfloor$ in their system models. Moreover, both algorithms are tailored for wireless embedded systems. They have low message complexity and use multicast to reduce the communication overhead, and they rely on neither low-level reliable transmission protocols, e.g. TCP, nor other complex primitives such as reliable broadcasting. Several application scenarios in the field of robotics and vehicular communication are investigated. For example, a use case of life-searching robots is introduced when explaining multi-value consensus and RATCHETA. In the end, a more complicated application in vehicular ad-hoc network named Maneuver Coordination service is introduced. A coordination protocol based on consensus is designed for Maneuver Coordination service, allowing a group of vehicles to reach agreement on their driving trajectories, which can improve traffic efficiency while keeping safety.

Zusammenfassung

Der Konsens ist ein grundlegendes Problem in verteilten Systemen. Heutzutage gibt es immer mehr kooperative und autonome Systeme, in denen verschiedene Teilnehmer koordinieren und zusamenarbeiten, um ein gemeinsames Ziel zu erreichen. Die meisten dieser Systeme erfordern eine hohe Fehlerresistenz – andernfalls könnte ein einzelner fehlerhafter Knoten das gesamte System unbrauchbar machen. Dies erfordert im Wesentlichen einen byzantinisch fehlertoleranten Konsens. Typischerweise können jedoch nur $\lfloor \frac{n-1}{3} \rfloor$ fehlerhafte Knoten in einer Gruppe von n Knoten toleriert werden, wenn das System partiell synchron ist. Diese Fehlertoleranzrate ist viel niedriger als in Absturz-Fehlertoleranz, wobei $\lfloor \frac{n-1}{2} \rfloor$ fehlerhafte Knoten toleriert werden können. Noch schlimmer, Systeme mit nur drei Knoten sind zu klein, um überhaupt einen einzelnen byzantinischen Knoten zu tolerieren. Da das byzantinische Fehlermodell, in dem Knoten beliebig fehlerhaft sein können, zu pessimistisch ist, wird in dieser Arbeit ein hybrides Fehlermodell betrachtet. In einem solchen Fehlermodell ist jeder Knoten mit einem kleinen vertrauenswürdigen Subsystem ausgestattet, das nur Crash-Fehler haben kann, während der Rest des Systems weiterhin byzantinisch sein kann. Durch die Nutzung des vertrauenswürdigen Subsystems werden zwei Konsens-Algorithmen entworfen: TRUSTED BEN-OR ist ein randomisierter Algorithmus, der den binären Konsens löst, und RATCHETA ist ein deterministischer mehrwertiger Konsensalgorithmus. Beide Algorithmen verwenden vertrauenswürdige monotone Zähler und verbessern die maximal tolerierbaren Fehler auf $\lfloor \frac{n-1}{2} \rfloor$ im asynchronen oder partiell synchronen System. Darüber hinaus sind beide Algorithmen auf drahtlose eingebettete Systeme zugeschnitten. Sie haben eine kleine Nachrichtenkomplexität und verwenden Multicast, um den Kommunikationsaufwand zu verringern. Sie verlassen sich weder auf zuverlässige Netzwerkprotokolle, z.B. TCP, noch andere Kommunikationsprimitive wie Reliable-Broadcast. Es werden verschiedene Anwendungsszenarien im Bereich Robotik und Fahrzeugkommunikation untersucht. Beispielsweise wird ein Anwendungsfall von Lebenssuchrobotern vorgestellt, wenn der mehrwertige Konsens und RATCHETA vorgestellt werden. Am Ende wird eine kompliziertere Anwendung im Fahrzeug-Ad-hoc-Netzwerk mit dem Namen Maneuver Coordination Service in Betracht gezogen. Koordinierungsprotokoll für den Maneuver Coordination Service wird entwickelt. Mit diesem Protokoll kann eine Gruppe von Fahrzeugen eine Einigung über ihre Fahrbahnen erzielen, wodurch die Verkehrseffizienz verbessert und gleichzeitig die Sicherheit gewährleistet werden kann.

Acknowledgements

I would like to thank all the people who gave me support and assistance during my research and my writing of this dissertation.

I would like to acknowledge all my colleagues. Johannes Behl, Bijun Li, Tobias Distler, Signe Rüsch and Ines Messadi gave me many advices and inspirations about the distributed consensus. Thank all of you for the active discussions. Also thank Stefan Brenner, David Goltzsche, Vasily Sartakov, Nico Weichbrodt, Manuel Nieke and Marcus Brandenburger for the discussion about the trusted execution. Thank Björn Cassens for the feedbacks about the embedded systems. Thank Bernd Lehmann and Keno Garlichs for the discussion about the vehicular communication. Martin Wegner and I worked together for a long time in the CCC project. That is an exiting research project and I appreciate the collaboration with all of the project members. Thank Anna Lux for hosting all the events of CCC.

I would like to thank my supervisor, Prof. Rüdiger Kapitza, who gives me supports all the time. Your expertise, advice and patience helps me overcome many difficulties and is invaluable for my PhD research. Thank Prof. Lars Wolf, Prof. Sándor Fekete and Prof. Rolf Ernst for the feedbacks of different scientific and technical topics, which help me extend the scope of my research.

I am grateful for the helps from all my friends. Special thank to David Tiong Kung Hai who encouraged me to start writing. Thank Pengfei, Kwai-Fan, Xinxin, Zijian, Qibei and Jisheng.

Last but not least, thank my parents for supporting me. Without your encourage, I could not have completed this dissertation.

Table of Contents

Title Page i

Abstract iii

Zusammenfassung v

List of Figures xiii

List of Tables xv

Abbreviations xix

1 Introduction 1

1.1 Motivation 1

1.2 Main Contributions 3

1.3 Structure of the Thesis 4

1.4 Related Publications 5

2 System Model 7

2.1 Processes and Hybrid Fault Model 7

2.2 A Brief Introduction to Trusted Execution Environment and ARM TrustZone 8

2.3 Timing Model and Communication Model 9

2.4 Problem Definition 11

2.5 Known Results of Fault-Tolerant Consensus 15

3 Randomized Binary Consensus 19

3.1 The Original Ben-Or's Algorithm 19

3.2 TRUSTED BEN-OR Algorithm 21

3.2.1 Algorithm Design 22

3.2.2 Message Authentication and Trusted Coin in the Trusted Subsystem 24

3.2.3 Message Certificate and Validation 25

3.3 Correctness proof 27

3.3.1 Agreement . . . 27
3.3.2 Termination . . . 28
3.3.3 Validity . . . 33
3.4 Discussion and Optimization . . . 33
3.4.1 When Randomization Meets a Strong Adversary . . . 33
3.4.2 Handling Omission Failures . . . 34
3.4.3 Decision Forwarding . . . 36
3.5 Evaluation . . . 36
3.5.1 Testbed and Methodology . . . 36
3.5.2 Experiment with Non-Faulty Processes and Omission Faults . . . 38
3.5.3 Experiment with Byzantine Processes and Omission Faults . . . 40
3.5.4 An Explanation About the Difference Between Odd and Even Group Sizes 42
3.6 Related Work . . . 43
3.7 Conclusion . . . 45

4 Deterministic Multi-Value Consensus 47
4.1 Multi-Value consensus . . . 47
4.2 Use Case Scenario and Preliminaries . . . 49
4.2.1 An Example: Post-Disaster Search and Rescue . . . 49
4.2.2 Hybrid Fault Tolerance: Preliminaries and Open Issues . . . 50
4.3 BiTrInc: the Trusted Counter Authentication . . . 51
4.4 RATCHETA Algorithm Design . . . 53
4.4.1 Unique Messages per Round . . . 55
4.4.2 Proposal Certificate . . . 59
4.4.3 Active Round-Forwarding and Late-Acknowledgment . . . 61
4.4.4 Implementation of Rounds . . . 64
4.5 Correctness Proof . . . 64
4.5.1 Agreement . . . 66
4.5.2 Termination . . . 68
4.5.3 Validity . . . 69
4.5.4 Limited Memory Usage and Message Size . . . 71
4.6 Optimization of Quorum Size . . . 72
4.6.1 Revised Definition of Quorum . . . 72
4.6.2 Revised Proof of Agreement . . . 73
4.6.3 Revised Proof of Termination . . . 75
4.6.4 Revised Proof of Validity and Limited Memory Usage / Message Size . . . 75
4.7 Other Optimization . . . 75
4.7.1 Decision Forwarding . . . 75
4.7.2 Skip the Round Change in the First Round (Weak Validity) . . . 76

4.7.3 Stubborn Broadcasting 76
4.8 Evaluation 76
4.8.1 Testbed and Methodology 76
4.8.2 Experiment with Non-Faulty Processes and Omission Faults 77
4.8.3 Experiment with Byzantine Processes and Omission Faults 80
4.9 Related Work 82
4.10 Conclusion 83

5 Consensus for Autonomous Maneuver Coordination **85**
5.1 Motivation 85
5.2 Related Work 86
5.3 Preliminary: Maneuver Coordination Service 87
5.4 Coordination Protocol of Negotiation 88
5.4.1 Consensus in Maneuver Coordination 88
5.4.2 Communication Pattern 90
5.4.3 Reach Agreement to Prevent Divergence 91
5.4.4 Consequences of Message Losses 92
5.4.5 Discussions and Challenges 94
5.5 Evaluation 95
5.5.1 Implementation 96
5.5.2 Results of the Speed Changes 98
5.5.3 Communication Latency 101
5.6 Conclusion 101

6 Summary **103**
6.1 Research Questions and Answers 103
6.2 Outlook 105

References **107**

List of Figures

2.1 A typical system architecture based on ARM TrustZone 10
2.2 Median validity definition 13
2.3 Proof of tight bound 16

3.1 Median of latency of TRUSTED BEN-OR in fault-free case in comparison to Turquois 39
3.2 Median of latency of TRUSTED BEN-OR in fault-free case 40
3.3 Median of latency of TRUSTED BEN-OR with Byzantine nodes in comparison to Turquois 41
3.4 Median of latency of TRUSTED BEN-OR with Byzantine nodes 42
3.5 The probability that a process successfully receives messages from a quorum with group size $n = 2k$ and $n = 2k + 1$. 44

4.1 Coordination among cooperative robots 49
4.2 Fault-free case of RATCHETA with three processes 58
4.3 Fault-free case of RATCHETA with four processes 59
4.4 Use counter h to prevent a faulty coordinator from in-round equivocation 60
4.5 Use counter ℓ to prevent a faulty process from concealing its history 62
4.6 A system ends up with all processes from different rounds 63
4.7 Late-acknowledgment to ensure the liveness 65
4.8 Communication pattern of RATCHETA 66
4.9 An example of the new definition of quorum 74
4.10 Median of latency of the original and optimized RATCHETA in fault-free case in comparison to Turquois 78
4.11 Median of latency of RATCHETA in fault-free case 79
4.12 Median of latency of the original and optimized RATCHETA with Byzantine nodes in comparison to Turquois 81
4.13 Median of latency of the optimized RATCHETA with Byzantine nodes 82

5.1 Communication pattern of the trajectory negotiation 90
5.2 Two scenarios where an agreement is required 93

5.3 The example to explain the necessity of the PROMISE message 94
5.4 The lane-join scenario 96
5.5 The speed of the three vehicles in the lane-join scenario 99
5.6 The accumulative speed of all the three vehicles 100

List of Tables

2.1 Known bounds and examples of different system characteristics and fault models. 16

3.1 Latency of authenticate/verify in ms 37

4.1 Notation used to explain BiTrInc 53

5.1 The time loss in seconds. 100

List of Algorithms

3.1 Ben-Or's algorithm 20
3.2 TRUSTED BEN-OR algorithm 23
4.1 RATCHETA algorithm 56
4.2 The function to verify a proposal certificate satisfies weak validity. 57
4.3 The function to verify a proposal certificate satisfies median validity (requires $n > 3f$). 57

Abbreviations

BFT Byzantine fault-tolerant 1

HMAC keyed-hash message authentication code 7

IoT Internet-of-things 1

OS operating system 8

SMR state machine replication 2

TEE Trusted execution environment 8

V2V vehicle-to-vehicle 81

1 Introduction

1.1 Motivation

Distributed systems are systems where different processes work cooperatively to achieve a common goal. Besides those clusters that are located in data centers and cloud, nowadays small devices also tend to become smarter and more cooperative. They can communicate with each other via wireless communication and can work together as a group, *known as cooperative wireless embedded systems*. Examples are Internet-of-things (IoT) [49], wireless sensor networks used for machine monitoring and automation in large factories [35], vehicular platooning to reduce energy consumption [45], unmanned aerial vehicle (UAV) swarms for life search and rescue missions [19] and even satellite swarms for geology research [57], just to name a few. In such scenarios, a distributed consensus is an essential building block to ensure coordinated actions in the cooperative groups. Unfortunately, to achieve consensus is not a trivial task. We could rely on a centralized coordinator, which however leads to a single point of failure. Any misbehavior of the coordinator would likely break the whole system. For example, if the coordinator crashes, the remaining participants will be in chaos or fall back to uncoordinated actions. Even worse, a malicious coordinator can mislead the whole group and deliberately cause inconsistency. The latter misbehavior is categorized into the so-called *Byzantine faults* [40], meaning that a process acts arbitrarily wrongly, including not only crashing but also working actively against its algorithm specification. Byzantine faults can be caused by software/hardware errors, sensor/actuator malfunctions, malicious attacks, etc. Accordingly, a Byzantine fault-tolerant (BFT) consensus in the distributed system would be desirable to guarantee that the system can stay operational despite a limited number of Byzantine faults.

Although resistant against arbitrary faults, BFT consensus protocols have several drawbacks, making them hard to be applied in real-life systems. The first concern is the low fault-tolerance rate. In a partially synchronous system, a total of n processes can only tolerate up to $\lfloor \frac{n-1}{3} \rfloor$ Byzantine processes [22]. This number is quite small compared to $\lfloor \frac{n-1}{2} \rfloor$ of its crash-fault tolerant counterpart, e. g. Paxos [39]. Secondly, a BFT consensus protocol usually has a high complexity in respect of the number of messages and communication rounds — leading to a big overhead in the execution. We take the PBFT [15] protocol as an example. In the fault-free case, it takes three rounds of one-to-all or all-to-all message exchange until termination. If the primary fails, at least two more rounds are required to run the so-called view-change sub-protocol to recover from a failure.

To tolerate Byzantine faults is difficult, because a faulty process is able to cheat in a way that is hard to be detected. In the example of the Byzantine generals' problem [40], a "two-faced" general can lie to two lieutenants by telling one lieutenant to attack whilst telling the other to retreat. It reminds us that a process cannot unconditionally trust the words of another process, but it has to check with the others to detect such two-faced lies, which causes extra communication rounds. In contrast to *detecting* such misbehaviors, we can also *prevent* it by letting each process be equipped with a specialized trusted subsystem. Whenever a Byzantine process wants to send a contradictory message against its previous words, the subsystem would refuse to authenticate that message. The subsystem must be so reliable that even a faulty host system cannot compromise the subsystem except for stopping its service. The idea to assume a more trustworthy subsystem among the less trusted parts is referred to as a *hybrid fault model* [70]. The trusted subsystem can protect the execution of functions inside it, and we can rely on the correctness of these functions. Only a minimum number of critical functions should be chosen to be protected, otherwise the implementation cost and the probability of errors will increase as the subsystem becomes more complex. In this thesis, we show that we can rule out those two-faced liars mentioned above by putting only a few functions inside the trusted subsystem. As a result, the fault-tolerant rate under this hybrid fault model can achieve $\lfloor \frac{n-1}{2} \rfloor$ in the partially synchronous system, which is the same as in crash fault-tolerance.

One of the most typical application scenarios of BFT consensus is the state machine replication (SMR) [63], where a server is replicated over different places. Each replica keeps a copy of the service state, and all replicas execute the operations from the clients' requests in the same order. For this purpose, the replicas have to rely on a consensus protocol to agree on the order of executions. The consensus protocol itself must be fault-tolerant as well. The idea of utilizing a trusted subsystem in consensus is also adopted by several SMR systems [16, 43, 71, 37, 6], to improve the fault-tolerant rate to $\lfloor \frac{n-1}{2} \rfloor$.

However, there is limited research focusing on the cooperative embedded systems. Although cooperative embedded systems share a similarity with SMR from the functionality aspect, they have their own characteristics. Firstly, it is more urgent to tolerate as many faulty processes as

possible in a cooperative system, because the system is distributed *inherently* according to the classification of Cachin et al. [13]. In contrast, an SMR is distributed *as an artificial*. The designers of an SMR system will firstly analyze and determine how many simultaneous faulty processes (the value f) should be considered, then decide how many replicas is required (the value n). If the f exceeds a threshold with respect to n, the system designer can introduce more replicas accordingly. However, this could be impractical in an inherent distributed system where there are already n processes. As a corner case, there is no $\lfloor \frac{n-1}{3} \rfloor$ fault-tolerant consensus protocols that can be applied in a group of only three participants. For example, the Swarm project [57] is composed of only three satellites. If people decide to make the system resilient against Byzantine faults someday, a traditional BFT consensus requiring $n = 3f + 1$ nodes is useless, while providing a fourth satellite only for the purpose of fault-tolerance is clearly out of range for cost reasons.

Secondly, the system characteristics are different between embedded systems and servers in data center or clouds. The most significant difference is the computational power, which limits the embedded systems to use a too complex protocol. Moreover, in dedicated application scenarios of embedded systems, some low-level communication protocols such as TCP could be missing. Therefore, a consensus protocol in those systems should have the minimum dependencies on them. For instance, the consensus has to deal with message omissions on its own if no reliable transmission communication protocol is provided.

Thirdly, the workflow and requirements for cooperative embedded systems are different from those in SMR. This can result in differences in design and optimization of the consensus protocol. One issue is the value validity. In SMR, the value to be agreed, i. e. the order to execute a command, is normally proposed by a dedicated coordinator (or primary). [1] In a cooperative system, however, each process can have its own proposal, e. g. from its sensor, which can influence the final decision. As a result, there must be certain constraints to the agreed value, otherwise faulty processes can mislead the whole group to make a wrong decision.

1.2 Main Contributions

This dissertation studies the BFT consensus problem in wireless embedded systems under the hybrid fault model. Two concrete consensus protocols are designed and evaluated. Both can tolerate up to $\lfloor \frac{n-1}{2} \rfloor$ faulty processes in a group of n in the asynchronous system or partially synchronous system respectively. Here we highlight some important contributions below.

TRUSTED BEN-OR is a randomized binary consensus protocol that allows processes to decide a value between 0 and 1. It can achieve consensus in a fully asynchronous system, because it utilizes randomization to bypass the theoretical impossibility, namely the FLP impossibility [31]. It might be the first hybrid fault-tolerant consensus protocol with a complete proof. Moreover,

[1] This is very similar to the original Byzantine generals problem [40], where only one general proposes to attack or retreat.

TRUSTED BEN-OR is proved to be resilient even against a strong adversary. We point out that several other works neglected some corner cases during the proofs of their randomized algorithms. We hope that the proof technique we adopted can draw attention of other researchers.

RATCHETA is a multi-value consensus protocol running in a partially synchronous system. It relies on a pair of trusted counters to force the faulty processes to stick to their words and prevent them from cheating. Compared to other solutions using only one counter [43, 71, 37], RATCHETA can guarantee limited memory usage and message size. The design and proof of the protocol also become simplified because of the use of two separate counters.

Both protocols are tailored for wireless embedded systems. They do not rely on low level reliable broadcast primitives nor reliable transmission protocols such as TCP. Neither do they explicitly detect or handle message omissions. Only a stubborn broadcast mechanism is required. The evaluation results also show that both protocols perform well under the existence of omission failures. [2] Moreover, peer-to-peer communication is needed as less as possible. Most messages are broadcast to all so that the common communication medium in a wireless network can be fully utilized.

A new definition of validity, namely the median validity in asynchronous system is proposed, as the classical definitions of validity do not apply to continuous value space, which is commonly found in sensor networks. We prove that this definition is a tight bound in asynchronous systems.

Finally, the Maneuver Coordination service is designed as a use case scenario to leverage distributed consensus. It helps autonomous vehicles spontaneously form a cooperative group and dynamically negotiate the driving trajectories. We have designed the communication pattern and analyzed the impacts of possible failures. Because of the consensus mechanism, no severe disagreement that threatens the safety can happen. The simulation shows that in the best case, the Maneuver Coordination service can provide a coordinated driving plan to the nearby vehicles, which is much more efficient than the default right-of-way rule.

1.3 Structure of the Thesis

The following chapters of this thesis are organized as follows:

Chapter 2 introduces the system model and the problem definition of distributed consensus. The asynchronous median validity is defined alongside with the problem definition. Meanwhile, the hybrid fault model is illustrated and an overview of trusted execution and the ARM TrustZone is given.

Chapter 3 details the randomized TRUSTED BEN-OR protocol. The focus is on the correctness proof under a strong adversary. We also discuss some corner cases such as the possible message

[2] Theoretically, none of them are resilient against omission failures. This is another tough task even in a fully synchronous system [61].

omission failures, and some optimization strategies. In the end, the evaluation results in a real distributed setting are shown.

Chapter 4 presents RATCHETA protocol. We firstly analyze two different malicious actions a faulty process can take, and how to prevent them with two monotonic counters. The evaluation results are also reported.

Chapter 5 describes the application of consensus in vehicular network and presents the Maneuver Coordination service. The communication protocol is explained, and the impact of potential failures is discussed. We test Maneuver Coordination service in a simulated environment and show the efficiency improvement on accumulated driving speeds.

Chapter 6 concludes the work of this thesis and suggest the direction of the future works.

1.4 Related Publications

[78] W. Xu and R. Kapitza. "RATCHETA: Memory-Bounded Hybrid Byzantine Consensus for Cooperative Embedded Systems". In: *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*. Oct. 2018, pp. 103–112. DOI: 10.1109/SRDS.2018.00021.

[79] W. Xu, M. Wegner, L. Wolf, and R. Kapitza. "Byzantine Agreement Service for Cooperative Wireless Embedded Systems". In: *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. June 2017, pp. 10–15. DOI: 10.1109/DSN-W.2017.45.

[80] W. Xu, A. Willecke, M. Wegner, L. Wolf, and R. Kapitza. "Autonomous Maneuver Coordination Via Vehicular Communication". In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. June 2019, pp. 70–77. DOI: 10.1109/DSN-W.2019.00022.

[81] Wenbo Xu, Signe Rüsch, Bijun Li, and Rüdiger Kapitza. "Hybrid Fault-Tolerant Consensus in Asynchronous and Wireless Embedded Systems". In: *22nd International Conference on Principles of Distributed Systems (OPODIS 2018)*. Ed. by Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira. Vol. 125. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 15:1–15:16. ISBN: 978-3-95977-098-9. DOI: 10.4230/LIPIcs.OPODIS.2018.15.

2

System Model

In this chapter we define the system model, including the processes, fault model, timing model and the definition of the distributed consensus problem.

2.1 Processes and Hybrid Fault Model

We consider a static group consisting of n processes, and every process has a unique ID $\{p_1, \ldots, p_n\}$ that are known to each other. Each process has an initial value, which is an external input to the consensus algorithm. We consider the following hybrid fault model. A process is called *correct* if it exactly follows the algorithm specification. Some processes, at most f, can be *faulty*. The faulty processes can crash, claim an incorrect value or even actively work against the algorithm — also known as *Byzantine faults*. They can also collude with each other. Basically, Byzantine processes can be arbitrarily faulty, but there are two exceptions. Firstly, Byzantine processes are unable to break the cryptographic mechanisms including the asymmetric digital signature and the symmetric key authentication such as keyed-hash message authentication code (HMAC). Neither do they know the secret key(s) of the correct processes. [1] Secondly, every process possesses a trusted subsystem that is tamper-proof and will always behave honestly. Even a malicious host process is unable to compromise its own trusted subsystem, except for stopping its service. The trusted subsystem can also restore its state after it restarts (fail-recover). In other words, in our hybrid fault model, a faulty process has a Byzantine part and a fail-recover part. In the next section, I will introduce how to build such a system in practice.

[1] A Byzantine process can reveal its secret key to others, but a correct process will never do this.

Remark 1 *In the rest of this thesis, n denotes the total number of processes, where f denotes the maximum number of faulty processes that can be tolerated. Note that f is just a number. When it says "f processes", it does not mean they are faulty.*

2.2 A Brief Introduction to Trusted Execution Environment and ARM TrustZone

There have been several approaches to building a trusted subsystem to prevent malicious attacks. In the following we list three common solutions:

- Secure Element (SE) or Trusted Platform Module (TPM): they are external hardware platforms that can store secure application codes and confidential data, such as cryptographic keys. They possess a dedicated microcontroller and storage that are separated from the main platform, so they provide a very high level of isolation and a root of trust. However, SE or TPM have two main drawbacks. Firstly, they have limited processing power because they do not use the CPU of the main platform, so they cannot execute too complex codes. Secondly, the applications running in the SE or TMP are fixed and lack of adaptability. Users cannot modify the already-installed trusted applications, nor install new applications. Thus, they are only suitable for some simple and common functions that are less likely to change after deployment. Cryptographic attestation is a good example here.
- Trusted execution environment (TEE): a TEE provides trusted application an isolated execution environment, which runs in parallel to the non-trusted components. Unlike SE or TPM, this isolation does not require extra chips or co-processors, but only utilizes hardware extensions of the main platform. Both the trusted and non-trusted parts are using the same CPU(s). Its level of isolation is not as high as the SE or TPM, but it allows users to develop customized trusted applications.
- Software virtualization: it provides software-level isolation of execution. The trusted code runs in an isolated compartment, e. g. a virtual machine or a container, that does not directly interfere with the non-trusted software. An example of software virtualization is the software-based *hypervisor* (or *virtual machine monitor*), which manages multiple virtual machines on the same hardware platform or operating system (OS). Even a privileged user of one virtual machine has no access to the resource of another virtual machine. In this way, resources are isolated and protected. Compared to the other two solutions above, this approach does not require specific hardware support, but is less secure. A malware can exploit the vulnerability of the hypervisor or the host OS running the hypervisor, thus compromise the whole system including the virtual machines.

In order to choose the correct technology, or a combination of them, one needs to consider different aspects, including the security requirement, attacker model, performance requirement,

cost and adaptability. Considering our use case scenarios, we choose TEE to realize the hybrid fault model, because on the one side, a high level of safety and security is required in some embedded systems such as robotics and vehicles, so software virtualization can hardly meet this requirement; on the other side, the protected functions should also be customizable, whereas SE and TPM cannot provide this feature. Moreover, SE and TPM can suffer performance degradation. If we implement a simple trusted monotonic counter as we will use in the next chapters, a TPM is much slower than a TEE as pointed out by Brandenburger et al. [11].

Specifically, we choose *ARM TrustZone* because of the popularity of ARM architecture among embedded systems. Here we give a brief introduction to ARM TrustZone. To achieve isolation, ARM TrustZone divides the software and hardware into two worlds: a *secure world* and a *normal world*. This division is in parallel to the privilege levels, which is another mechanism to ensure access control and separation. Namely, in either world there can be different privilege levels, but a high-privileged process from the normal world cannot directly access any process in the secure world, even if the latter has a lower privilege. As a result, ARM TrustZone provides an isolation to protect the data and the execution in the secure world, including the memory regions, Translation Lookaside Buffers (TLBs), caches, system controls, etc. This isolation is a combination of the physical and virtual mechanisms. On the one hand, a special bit of the main system bus can identify each read/write transaction as secure or non-secure. Thus, resources that belong to secure world cannot be accessed from the normal world. On the other hand, both the normal world and secure world run on the same processor in a time-sliced manner.

Figure 2.1 presents a common system architecture based on ARM TrustZone. In the normal world is the ordinary OS, or *rich OS*. In the secure world, a secure OS with only limited and essential functions is running. As mentioned above, processes from the normal world cannot directly access the secure world resources, but they can communicate with the latter via a secure monitor. As to more details such as the context switch between the two worlds, Ngabonziza et al. have given a comprehensive description [55], so we omit the discussions here.

2.3 Timing Model and Communication Model

Two timing models will be considered in this thesis. The first is the *asynchronous* model, where there is no bound on the processing and communication delay. In this situation, if a recipient is waiting for a message which does not come, it cannot tell whether this message is delayed, or the sender is faulty and did not send it at all, so the recipient will not wait for the message forever. Thus, if a recipient is waiting for messages from n processes, and it knows at most f of them can be faulty, it must take an action after it receives $n-f$ messages, but not wait for the rest f messages.

The asynchronous model represents a pessimistic yet realistic system. In such a model, any deterministic consensus algorithm cannot tolerate even a single crash fault while still guarantee

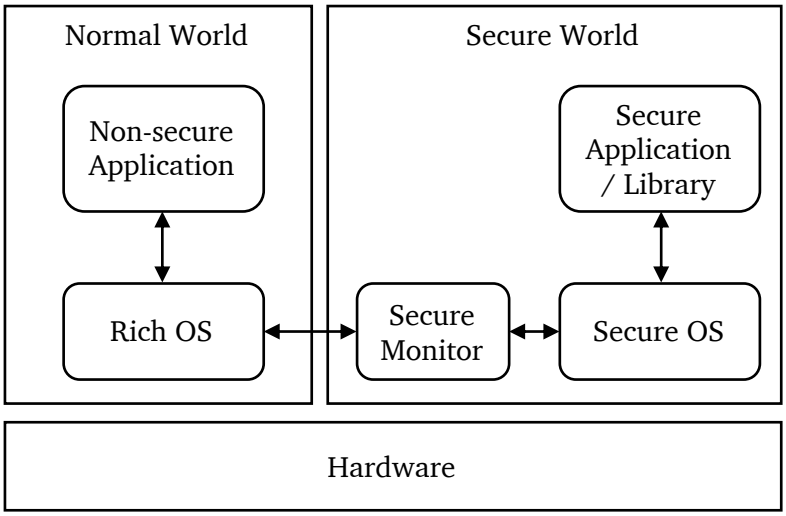


Figure 2.1: ARM TrustZone overview.

both the safety and progress at the same time [31]. To bypass this impossibility, either we can resort to randomization, or we have to enhance the asynchronous timing model as we will see right now.

The second timing model we considered is the *partially synchronous* model [22]. It assumes that the system starts from an asynchronous state, but eventually there is a *Global Stabilization Time* (GST), after which the system becomes synchronous, i. e. message delivery and processing delay becomes bounded. However, any process at anytime does not know when this GST will come, nor whether it has already come.

For the purpose of presentation, we adopt the automata abstraction [13, pp.20-22], where each process is modeled as an automaton. An automaton runs step by step according to the specification of an algorithm. Assume there is a *global wall-clock*, which is nevertheless not available to the processes, and a virtual *scheduler*. At each time (a clock tick) t, the scheduler chooses a process to take a step, e. g. to deliver a message, or to execute a line of the algorithm, etc. We call each step an *event*. For simplicity, we assume that at each clock tick only one event can take place.

We firstly assume a reliable communication so that every message transmitted between correct processes is eventually delivered. Later we will discuss the message omission issue in the following chapters. Messages are not encrypted, but are authenticated to ensure integrity and authenticity. Sometimes a message also requires a digital signature for non-repudiation. Later we will show that we can utilize the more efficient symmetric authentication code rather than asymmetric digital signature to achieve non-repudiation with the help of the trusted subsystem.

2.4 Problem Definition

In a distributed consensus problem, each process p_i takes an input from the value space $\mathcal{D}$ as its initial value and decides a value $v_i \in \mathcal{D}$ as its output.

Definition 1 *An algorithm solving consensus problem is correct, only if it satisfies the following three properties:*

- ***Agreement****: No two correct processes decide differently.*
- ***Termination****: Every correct process eventually decides.*
- ***Validity****: The decided value must meet certain validity criteria (will be discussed below).*

Different definitions of the validity. The first two requirements correspond to the safety and liveness property of the algorithm. The validity is required to preclude some obviously trivial solutions and makes the algorithm really useful. Without the validity requirement, we can simply implement an algorithm by letting every process decide 0, no matter what inputs they do have.

There are several different definitions regarding the value validity. Ideally, we hope a strict validity can be fulfilled so that the decision is always correct:

Definition 2 ***Strict validity***[2]*: the decided value should be proposed by a correct process.*

This is the strictest type of validity. However, to achieve this, the total number of processes should be greater than $f \cdot |\mathcal{D}|$ even in a synchronous system [54], where $\mathcal{D}$ is the value space. Thus, if the value space becomes too large, it is very difficult to achieve the strict validity.

A relaxed definition named *strong validity* is commonly used in the literature. It is defined as:

Definition 3 ***Strong validity****: If all the correct processes propose the same value v, they must also decide v.*

To achieve strong validity, there is also a requirement with respect to n and f :

Lemma 1 *There is no consensus algorithm that can achieve strong validity in an (partially) asynchronous system, unless $n \geq 3f + 1$.*

Proof. We follow the proof sketch of Lamport et al. [40]. By contradiction, we assume such a consensus algorithm with $n \leq 3f$ exists. For simplicity we only consider the case $n = 3f$. Consider the following two cases.

Case 1: There are f correct processes have the initial value 0, f correct processes have 1, and the rest f faulty processes crash immediately after the algorithm starts. That means, actually only

[2]Different literatures may adopt different terminologies for the validity definitions. For instance, Neiger calls this property as "strong validity", and calls the Definition 3 as "validity".

f processes with 0 and f processes with 1 are running the algorithm. If the algorithm terminates, all the correct processes must decide the same value. Without loss of generality assume they decide 0.

Case 2: All the $3f$ processes have the same initial value 1. There are f correct processes whose messages are extremely slow because of the asynchrony. Meanwhile, the f faulty processes pretend that their initial values are 0. This case is actually the same as Case 1, so the decision must also be 0. However, 0 is proposed by none of the correct processes. This leads to a contradiction to the definition of the strong validity. ∎

The impossibility is due to the fact that a slow process cannot be distinguished from a crashed process in an asynchronous system. Considering $n \geq 3f + 1$ is still a strong requirement, we can further relax the validity definition, for example the *weak validity* in a system where $n \leq 3f$:

Definition 4 ***Weak validity****: the decided value must be proposed by some process.*

That means, a decision may be merely proposed by a faulty process. Theoretically, this gives the malicious adversary an opportunity to compromise the system by letting processes decide on an arbitrary value. Nevertheless, in certain application scenarios, this vulnerability can be fixed via application-dependent knowledge. In those applications, a faulty value can be detected and rejected. For example, in a leader election algorithm with known identities, a valid proposal must be the identity of one participating process.

Note that albeit their names, the strong validity is not necessarily "stronger" than the weak validity. The strong validity does not specify what value is valid if the correct processes propose different values. The decision is even not required to be proposed by any process, which is however required by the weak validity.

Median validity. We may encounter some application scenarios where both weak validity and strong validity cannot meet our requirements. For example in a sensor value consensus, even the values of correct sensors could differ from each other because of the instrument errors. For weak validity, a single faulty process can make the whole group to decide an outlier value. For strong validity, when correct processes do not propose the same value, there is no specification about the decision. Actually, what we expect is that the decision lies within some certain bounds, no matter it is proposed by a correct or faulty process. Stolz and Wattenhofer proposed the definition of *median validity* [66], which requires the decision to be close to the median of all correct proposals. This implies that the values should be comparable to each other. Their original definition only applies to the synchronous system. We modify the definition to adapt to the asynchronous system as following:

Definition 5 *Assume $n \geq 3f + 1$. Let n_c denote the number of the correct processes during runtime (yet not known to the algorithm). These n_c correct initial values are sorted in the array*

$Sorted\_Correct$ in an ascending order. Denote $c := \lfloor \frac{n_c - 1}{2} \rfloor$, namely the index of the median of $Sorted\_Correct$. [3] *A decision v fulfills* median validity*, if*

$$Sorted\_Correct[c - f] \leq v \leq Sorted\_Correct[c + f] \tag{2.1}$$

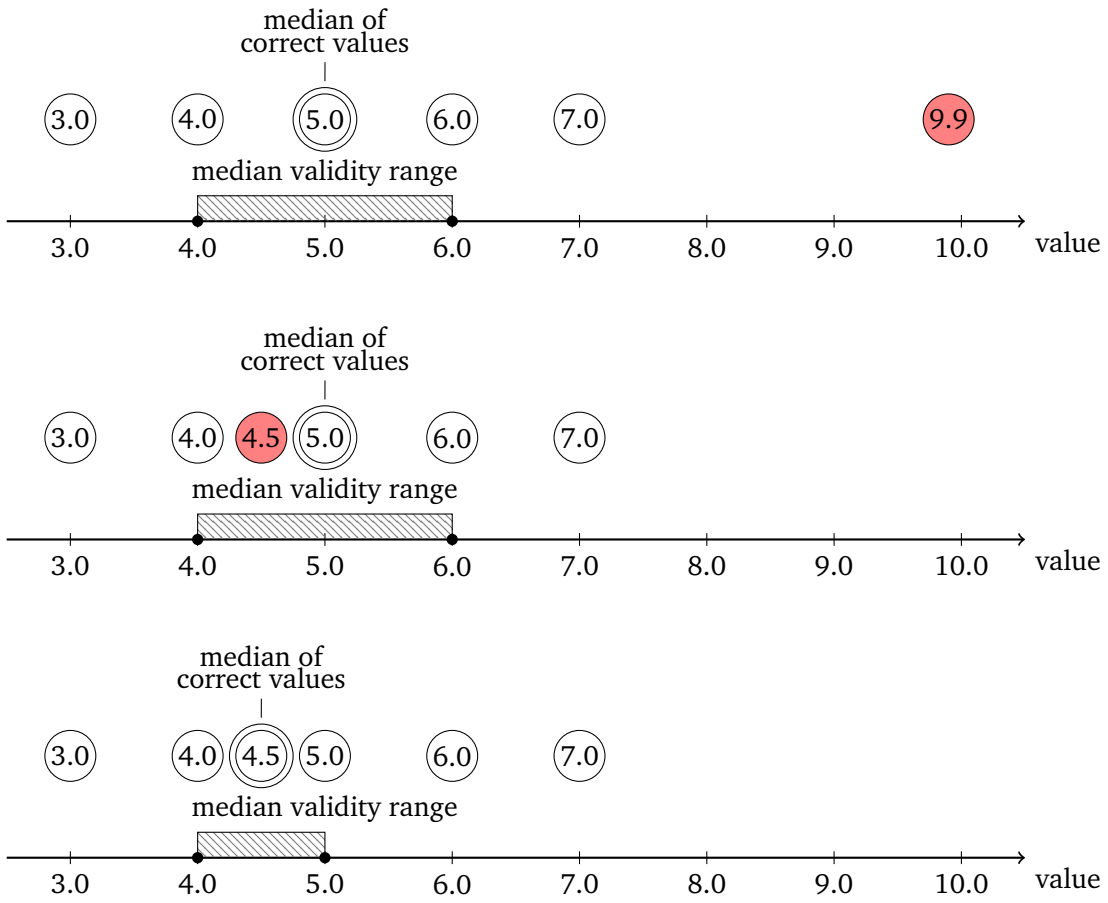


Figure 2.2: An illustration of median validity with $n = 6$ and $f = 1$. The red circles represent faulty processes. Note that in the third case, there is actually no faulty process during runtime.

Note that $n_c \geq n - f$, because f is the maximum number of the tolerable faulty processes, but does not mean that there must be f faulty processes. This definition can be illustrated as the example in Figure 2.2 with $n = 6$ and $f = 1$. Here we show three cases. In the first case there is an outlier value, whereas in the second case the faulty process proposes a value that "looks good". However, the range of a valid value are the same in both cases, no matter which value the faulty process proposes, because the range only depends on the correct values according to the definition. In the third case, there is actually no faulty process during runtime, so the median of the correct values and the validity range change accordingly.

[3] We choose the smaller index close to the middle as median if the array length is even.

Remark 2 *Given the assumption of $n \geq 3f+1$, the index $(c-f)$ or $(c+f)$ cannot go out of bounds. Actually, this assumption is indispensable, if we require the decided value to be "reasonable". Otherwise, suppose $n = 3f$. Similarly to Lemma 1, we can prove that the decision can be based on only f correct values and f faulty values, while the rest f processes are too slow. The f faulty processes can collude and claim an extreme outlier value to arbitrarily influence the decision. There is no way to distinguish which values are the real outliers without external information. In this case, it is pointless to quantify a validity range.*

Obviously, if all correct processes propose the same value, then this value is the only valid one, so median validity implies strong validity. However, it is orthogonal to the strict validity. On the one hand, a decision satisfying median validity can also be from a faulty process. However, in order to let its proposal to be decided, a faulty process has to choose a good enough value within a certain range, otherwise it will be ignored, as shown in the first two cases of Figure 2.2. On the other hand, the strict validity does not imply median validity, either. Even if a value is proposed by a correct process, it can be regarded as invalid. This is shown in the third case of Figure 2.2. The value 3.0 and 7.0 are too far away from the median and are not median valid here, although they are from correct processes.

The Definition 5 is relaxed compared to the original definition [66], because we consider (partially) asynchronous systems, but we can prove that this is the tight bound in all (partially) asynchronous systems:

Lemma 2 *Given $n \geq 3f+1$, no deterministic consensus algorithm in an asynchronous system can guarantee that the agreed value v always satisfies either*

$$Sorted\_Correct[c-f] < v \leq Sorted\_Correct[c+f] \tag{2.2}$$

or

$$Sorted\_Correct[c-f] \leq v < Sorted\_Correct[c+f] \tag{2.3}$$

Proof. We proof by contradiction. Assume there is a consensus algorithm that can tolerate up to f Byzantine processes and satisfy the inequality 2.2. Consider the following three configurations where there are indeed f Byzantine processes during runtime:

1. The $n-f$ correct processes have the initial values $V_1 = \{f+1, f+2, \ldots, n\}$ while all f faulty processes are crashed. If the algorithm can terminate, denote the final decision of correct processes as v.

2. Now consider the configuration in which the f faulty processes have (or just claim to have) the initial values $\{n-f+1, n-f+2, \ldots, n\}$, and then exactly follow the consensus algorithm. The correct values are $V_2 = \{1, 2, \ldots, n-f\}$ and are divide into two groups. If one correct process has the same initial value as the first configuration, namely between $f+1$ and

$n-f$, it belongs to the fast group and has the same transmission and processing speed as the first configuration. All the other ones between 1 and f are so slow that they cannot be distinguished from crashed ones. As a result, the algorithm will treat this configuration the same as the first one, and will decide the same value v satisfying the inequality 2.2

$$V_2[c-f\,] < v \leq V_2[c+f\,] \text{ where } c = \lfloor \frac{n-f-1}{2} \rfloor \tag{2.4}$$

3. Now the correct values are $V_3 = \{2f+1, 2f+2, \ldots, n+f\}$. The first half processes, namely from $2f+1$ to n are fast, and the rest from $n+1$ to $n+f$ are extremely slow. The (claimed) initial values of the faulty processes are $\{f+1, f+2, \ldots, 2f\}$ and run the algorithm honestly. Still the algorithm cannot distinguish this configuration from the other two since the actively participating processes behaves all the same.

   As a result the same value v is decided satisfying

$$V_3[c-f\,] < v \leq V_3[c+f\,] \text{ where } c = \lfloor \frac{n-f-1}{2} \rfloor \tag{2.5}$$

Clearly for any index i from 0 to $n-f-1$, $V_2[i] = i+1$ and $V_3[i] = i+2f+1$. So there must be $V_3[c-f\,] = c+f+1 < v \leq V_2[c+f\,] = c+f+1$, which leads to a contradiction. The same contradiction occurs for inequality 2.3. ∎

The proof can be illustrated as in Figure 2.3. The dashed circle is correct but slow process, and the red circle is a faulty process. In both cases, the actually participating values are $\{4.0, 5.0, 6.0, 7.0, 8.0\}$, so they can possibly decide the same value. However, the ranges bounded by inequality 2.2 or 2.3 in the two cases do not intersect at all.

2.5 Known Results of Fault-Tolerant Consensus

We can classify different consensus algorithms according to: 1) timing model, 2) fault model and 3) determinism. Table 2.1 lists the proved bounds between n and f and the notable examples correspondingly.

The impossibility of deterministic fault-tolerant consensus in asynchronous system is proved by Fischer et al. [31]. Furthermore, we can easily prove the $2f+1$ bound for crash faults by contradiction. Examples of such protocols include Paxos [39] for partially synchronous system and randomized Ben-Or's algorithm [7] for asynchronous system. As to BFT consensus, Dwork et al. proved that $n \geq 3f+1$ is a sufficient and necessary condition. A widely applied deterministic BFT algorithm reaching this bound in a partially synchronous system is PBFT [15]. Examples of randomized BFT protocols for asynchronous system include Bracha-Toueg's algorithm [10] and Turquois [51].

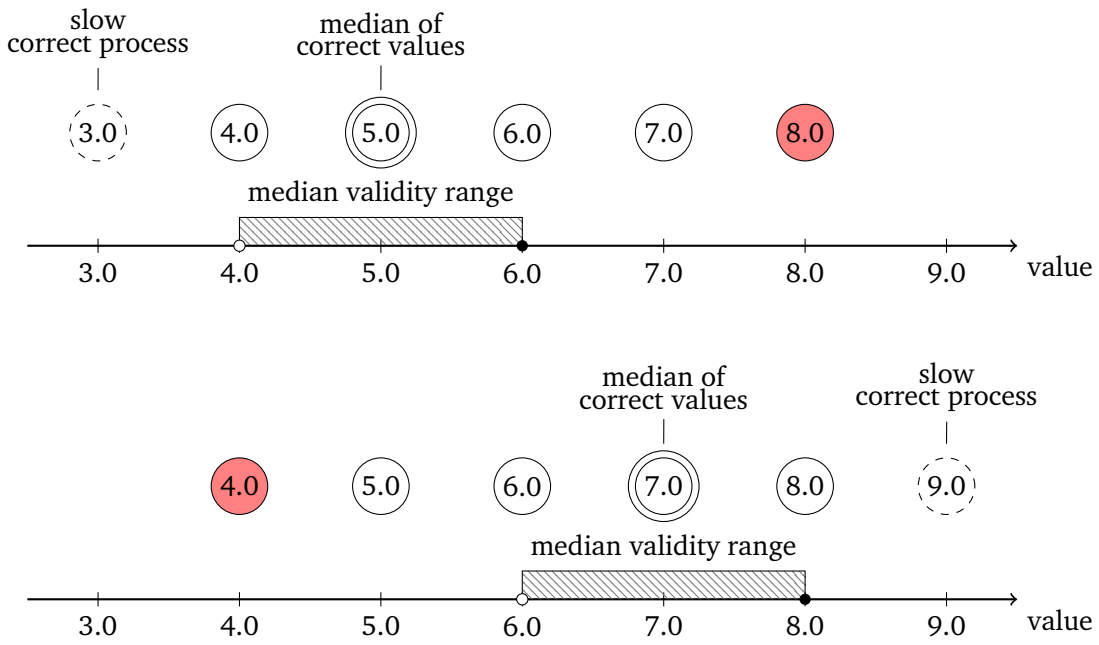


Figure 2.3: Proof of tight bound with contradiction.

		Crash fault	Byzantine fault	Hybrid fault
Deterministic	Fully asynchronous	Unsolvable with $f \geq 1$ [31]		
	Partially synchronous	$n \geq 2f + 1$ Paxos [39]	$n \geq 3f + 1$ PBFT [15]	$n \geq 2f + 1$ CheapBFT [37], Hybster [6]
Randomized	Fully asynchronous	$n \geq 2f + 1$ Ben-Or [7]	$n \geq 3f + 1$ Bracha-Toueg [10], Turquois [51]	N.A. (this dissertation)

Table 2.1: Known bounds and examples of different system characteristics and fault models.

However, the classification of fault model with just crash-only and Byzantine is too coarse-grained. Either a process will work correctly except for crashing, or it will be arbitrarily faulty, and there is no space between the two extremes. The hybrid fault model is introduced to fill the gap, and it is the focus of this dissertation. Under this fault model, faulty participants can still do more than just crash, but they are also restrained to certain extent. Apparently, its bound must lie somewhere between the other two fault models, depending on how the faulty behavior is restrained, because things cannot get better than in crash fault model, whilst not worse than in Byzantine fault model. Thus, we aim at assuming as less restrictions on the faulty behaviors as possible, whilst achieving the same bounds as in the crash-only fault model. For deterministic protocols in partially synchronous system, there are several protocols achieving $n \geq 2f + 1$ by relying on a relatively small trusted subsystem [43, 71, 37, 6], but we are not

aware of any randomized protocols for fully asynchronous systems. The latter will be covered in this dissertation.

3

Randomized Binary Consensus

As discussed before, there is no deterministic algorithm that can solve consensus problem while guaranteeing both the agreement and termination in an asynchronous system [31]. However, this impossibility can be bypassed via randomization. In a randomized algorithm, every process has access to a source of randomness, so the execution of a randomized algorithm is non-deterministic. In this chapter, we will introduce TRUSTED BEN-OR, a randomized binary consensus algorithm based on a hybrid fault model. It can tolerate up to $f \leq \lfloor \frac{n-1}{2} \rfloor$ faulty processes in an asynchronous system.

3.1 The Original Ben-Or's Algorithm

As the name of TRUSTED BEN-OR suggests, it is inspired by the Ben-Or's algorithm [7]. As a start point, we firstly introduce the original Ben-Or's algorithm. It is a simple and elegant algorithm that can solve randomized consensus. The algorithm can only tolerate crash fault, and assumes that a majority of processes are correct ($f \leq \lfloor \frac{n-1}{2} \rfloor$).

The Ben-Or's algorithm is listed in Algorithm 3.1. The algorithm is easy to follow. It consists of consecutive asynchronous rounds. Each round has a round number and can be divided into two phases: P-phase (short for proposing phase) and V-phase (voting phase). In the P-phase, a process simply broadcasts its current value as a proposal and collects proposals from a majority, including itself. In the V-phase, a process votes for a value. If the process receives the same proposal value from a majority, it will vote for that value. Otherwise, namely it has seen different proposals from a majority, it will vote for a default value $\perp$. After that, the process waits again to collect votes from a majority of processes. If the process receives more than f votes with the same value, which is not $\perp$, it can decide that value. No matter decided or not, the process updates its value

Algorithm 3.1: Ben-Or's algorithm

```
1   / Executed by process p /
2   v ← the initial value of process p
3   φ ← 0

5   loop forever:
6       φ ← φ + 1
7       broadcast ⟨PR, φ, p, v⟩                                  / P—phase /
8       wait for n − f ⟨PR, φ, i, v_i⟩ messages from different processes

10      if ≥ ⌈(n+1)/2⌉ messages carry the same value v_i = v′:   / V—phase /
11          broadcast ⟨VO, φ, p, v′⟩
12      else:
13          broadcast ⟨VO, φ, p, ⊥⟩
14      wait for n − f ⟨VO, φ, i, v_i⟩ messages from different processes

16                                                               / Decide and update /
17      if not decided and ≥ f + 1 messages carry the same value v ≠ ⊥:
18          decide v
19      if received at least one ⟨VO, φ, i, v_i⟩ with v_i ≠ ⊥:
20          v ← v_i
21      else:
22          v ← coin()
```

according to the following rule: if there is at least one vote carrying a non-$\perp$ value, the process updates its value to that one; otherwise, the process will toss a coin to randomly get a new value.

Is has been proved that Ben-Or's algorithm can work in a fully asynchronous system with a strong adversary [3]. It fulfills the following properties:

- **Agreement**: No two processes decide differently.
- **Termination**: Every correct process decides with probability 1.
- **Validity**: If a correct process decides v, v must be proposed by at least one process.

Note that the definition of termination is weaker than Definition 1 since "with probability 1" does not mean it must happen. The reason is that Ben-Or's algorithm can only terminate if every process who randomly tosses a coin gets a very "lucky" value at some round. If there are unlimited rounds, the probability that such a lucky coincidence happens will converge to 1. This analysis also answers the question why Ben-Or's algorithm is designed for binary consensus. Theoretically, it can also solve multi-value consensus. However, when the value domain becomes too large, the probability that every process happens to get its lucky value will be very small, so it is hard for the algorithm to terminate. Despite this limitation, binary consensus is important because it can serve as a base component, upon which multi-value consensus can be built [17, 50].

3.2 TRUSTED BEN-OR Algorithm

Inspired by Ben-Or's algorithm, we have designed TRUSTED BEN-OR, a randomized binary consensus algorithm based on a hybrid fault model that can tolerate up to $f \leq \lfloor \frac{n-1}{2} \rfloor$ faulty processes (namely $n \geq 2f + 1$). Its correctness criteria are defined as follows:

Definition 6 *Correctness of* TRUSTED BEN-OR.

- ***Agreement*** *No two processes decide differently.*
- ***Termination****: Every correct process decides with probability 1.*
- ***Validity****: If a correct process decides v, v must be proposed by at least $\lceil \frac{n-f}{2} \rceil$ processes.*

The validity definition implies the weak validity (Definition 4), but can be sometimes stronger than that, depending on the value of f and n. If $n \geq 3f + 1$, then $\lceil \frac{n-f}{2} \rceil > f$, so the decided value must be proposed by at least one correct process, which implies the strict validity as well as the strong validity (Definition 2 and 3). Otherwise, namely $2f + 1 \leq n < 3f + 1$, it degrades to the weak validity.

Before we take a closer look at it, we firstly highlight several contributions of the TRUSTED BEN-OR algorithm:

- TRUSTED BEN-OR is designed for fully asynchronous systems, namely messages can be arbitrarily delayed and reordered.
- The algorithm is resilient against a *strong adversary*, which means that the adversary can inspect the state of every process and message, and can coordinate all faulty processes and arbitrarily reorder message deliveries including messages from correct processes. We provide a correctness proof, which is the first complete proof under an asynchronous system with hybrid fault model and strong adversary to our knowledge.
- TRUSTED BEN-OR is tailored for wireless embedded systems for its simplicity and low complexity. Every message is sent via broadcast to make full use of the transmission medium. The communication does not require encrypted ciphertext, nor complex communication primitives such as reliable broadcast, nor TCP-like protocols that could be unavailable in certain application domains. Because of the trusted subsystem, the message authentication can use symmetric encryption, which is more efficient than asymmetric digital signatures.
- We implement TRUSTED BEN-OR and evaluate it in a real wireless ad hoc environment instead of in a pure simulation. The results are promising compared to Turquois [51], another well-known wireless ad hoc BFT consensus algorithm.
- We discuss some common issues regarding the termination of randomized BFT consensus algorithms, and point out that some algorithms might not be able to terminate in a strong adversary model.

3.2.1 Algorithm Design

When Byzantine faults exist, the correctness of the original Ben-Or's algorithm (Algorithm 3.1) does not hold anymore due to the following reasons:

- A Byzantine process may send a wrong value. For example, the faulty process can vote for 1 or $\perp$ in the V-phase but it actually should vote for 0.
- A Byzantine process may send contradictory values to different recipients during a broadcast. This behavior is referred to as *equivocation*. Assuming by the end of the V-phase, a faulty process has received $n-f$ messages voting for $\perp$, and f message voting for 0. Now this process has two options: either it chooses the $n-f$ $\perp$ to randomly get a new value and propose it in the next round, or it picks some 0-votes together with other $\perp$-votes to deterministically get a value 0. Both options are eligible, but the faulty process can equivocate from here.
- The termination relies on randomization, but a Byzantine process can break this by deterministically choosing a value instead of resorting to the random source. To illustrate this, we can consider a simple case of three processes p_1, p_2 and p_B. The first two processes are correct and they start with the same initial value. The Byzantine process p_B has a different initial value and follows Algorithm 3.1 exactly, except that it can manipulate the result of the random coin. Because both 0 and 1 exist at the beginning of the P-phase, the strong adversary can reorder the message delivery so that every process will receive different proposals, leading all of them to vote for $\perp$. As a result, every process will toss a coin by the end of the V-phase to randomly update its value. Even if the two correct processes happen to get the same value, p_B with the help of the strong adversary can know that result and deliberately get the opposite value. Eventually, the three processes enter a new round with different values again, and this procedure can repeat forever.

In order to overcome these difficulties, we introduce three mechanisms: *message certificate*, *message authentication* and *trusted coin*. A certificate can prove the correctness of a message and will be explained in Section 3.2.3. A message authentication is created by a trusted subsystem and can prevent equivocation. Trusted coin is an unbiased random number generator also protected by trusted subsystem. It is used together with the message authentication to guarantee the true randomness. These two mechanisms are detailed in Section 3.2.2.

Without formally defining the above-mentioned mechanisms, we firstly list the pseudo code of TRUSTED BEN-OR in Algorithm 3.2. The structure is similar to the original Ben-Or's algorithm, so we highlight the following differences between them.

Firstly, every message is authenticated and certificated. Upon receiving a message, a process has to check its validity. A message is called *valid* only if both its authentication and certificate are correct. How exactly it works will be explained in the following subsections.

Algorithm 3.2: TRUSTED BEN-OR algorithm

/ Executed by process p /
/ `authenticate`(m,u) and `authenticate_with_coin`(m,u) are trusted functions and will be explained in Section 3.2.2 /

$v_D \leftarrow$ the initial value of process p / Current value /
$\phi \leftarrow 0$ / Round number /
$flag \leftarrow$ *D-GET* / Whether the current value is deterministically or randomly got /

`authenticate`$(\langle INIT, \phi, p, v_D\rangle, 0)$ and broadcast it / The init round /
wait for $n-f$ valid $\langle INIT, \phi, i, v_i\rangle$ messages from different processes
if $\geq \lceil \frac{n-f}{2} \rceil$ messages have value 0
    $v_D \leftarrow 0$
else
    $v_D \leftarrow 1$

loop forever:
    $\phi \leftarrow \phi + 1$
    if $flag =$ *D-GET* / P—phase /
        `authenticate`$(\langle PR, \phi, p, v_D, \text{D-GET}\rangle, [\phi|0])$ and broadcast it with certificate
    else
        `authenticate_with_coin`$(\langle PR, \phi, p, \square, \text{R-GET}\rangle, [\phi|0])$ and broadcast it with certificate
    wait for $n-f$ valid $\langle PR, \phi, i, v_i, flag_i\rangle$ messages from different processes
    if $\geq \lceil \frac{n+1}{2} \rceil$ messages carry the same value v: / V—phase /
        `authenticate`$(\langle VO, \phi, p, v\rangle, [\phi|1])$ and broadcast it with certificate
    else:
        `authenticate`$(\langle VO, \phi, p, \perp\rangle, [\phi|1])$ and broadcast it with certificate
    wait for $n-f$ valid $\langle VO, \phi, i, v_i\rangle$ messages from different processes
    if not decided and $\geq \lceil \frac{n+1}{2} \rceil$ messages carry the same value $v \neq \perp$: / Decide and update /
        decide v
    if received at least one $\langle VO, \phi, i, v_i\rangle$ with $v_i \neq \perp$:
        $v_D \leftarrow v_i$
        $flag \leftarrow$ *D-GET*
    else:
        $flag \leftarrow$ *R-GET*

Secondly, there is an initialization round (line 8-13), in which each process broadcasts its initial value and picks the majority from the received $n-f$ ones to start round 1. As a result, if a process claims to have the value v in round 1, there must be at least $\lceil \frac{n-f}{2} \rceil$ processes that have proposed v in the initialization round. Moreover, this process can also prove this information to the others by piggybacking all these signed $INIT$-messages with value v as a certificate (Section 3.2.3). As we will see later, this property is necessary to prove the validity according to Definition 6.

Thirdly, each process must keep a flag indicating whether its value is taken deterministically (D-GET) or from a coin flip (R-GET). The flag must also be included in every *P-message* (the message sent in P-phase). This flag is necessary, because we will see later that a D-GET message has a different certificate compared to an R-GET message.

3.2.2 Message Authentication and Trusted Coin in the Trusted Subsystem

As mentioned above, one critically harmful Byzantine fault is equivocation, i. e. sending inconsistent messages to different recipients in a broadcast. Equivocation can be prevented by reliable broadcast [9], which takes several communication rounds and requires $n > 3f$. Thanks to the trusted subsystem, the equivocation can also be avoided by simply using a strict monotonic counter inside each process. Every message must be authenticated together with a counter value. After each authentication, the counter value increments and cannot be set back, so the same counter value cannot be used to authenticate two different messages. Levin et al. have provided more technical details about how this monotonic counter can be implemented [43]. We follow this idea and focus on how to use the counter in the consensus algorithm.

The algorithm also requires a random bit (line 20), so an unbiased trusted coin is placed inside the trusted subsystem. This prevents a Byzantine process from arbitrarily manipulating the result of the coin tossing. However, using the trusted coin alone is pointless. Without further measures, a Byzantine process can still repeatedly toss the trusted coin until it obtains its desired result, so the trusted coin should be used together with the monotonic counter. Whenever a random number is required, the coin tossing and the counter authentication should be integrated as an atomic operation. This ensures that the Byzantine process has only one chance to toss the coin with a specific counter value, preventing it from manipulating the random result.

The trusted subsystem maintains a unique identifier uid, a monotonically increasing counter value u, and necessary secrete key(s) to calculate message authentication codes. The uid is uniquely mapped to the process ID, so that a faulty process cannot forge the authentication in the name of others. The keys cannot be disclosed to the non-trusted part including the host system. The trusted subsystem provides the following APIs:

- `authenticate`(m,u): This function invocation takes a message m and a counter value u. It requires u greater than its last accepted value, otherwise the invocation is rejected. If called successfully, it will generate an authentication code based on $m||uid||u$ and append it to m. Meanwhile, the monotonic counter value is updated to u.

- `authenticate_with_coin`(m,u): This invocation is similar to the previous one. The only difference is that m should provide a field to receive a random bit. The trusted subsystem firstly generates a random bit and fills that field in m, then calculates the authentication code based on $m||uid||u$ and append it to m. The monotonic counter value is also updated to u. As shown in Algorithm 3.2 at line 20, we write m as $\langle PR, \phi, p, \square, R\text{-}GET \rangle$ where $\square$ is used to receive that random bit.

- `verify`(m, uid, u): It checks whether the authentication code appended to m is correctly generated based on $m||uid||u$ by the trusted subsystem uid.

Now that each message is guaranteed bound to a unique counter value, the remaining question is how to assign the corresponding counter value to every message, so that the recipients know which value they are expecting to authenticate the message and then know how to verify it. According to Algorithm 3.2, each process broadcasts exactly two messages in every round except for the initialization round, so a simple idea is to use the round number to define the counter value. More specifically, the P-message of round ϕ can be mapped to $u = 2\phi$, or equivalently $[\phi|0]$ where "|" is the separation of the least significant bit and higher bits, while the V-message can be mapped to $u = 2\phi + 1 = [\phi|1]$. The $INIT$ message is authenticated with counter value 0. In this way, every message is uniquely mapped to a counter value, and this value can be directly inferred from the round number and the message type. Meanwhile, when a correct process follows the algorithm specification, it will use the counter values in a monotonically increasing order: $0 \rightarrow 2 \rightarrow 3 \rightarrow 4\ldots$ (note that 1 is not used), so it can successfully authenticate every message.

Besides the monotonic counter to prevent equivocation and the trusted coin to ensure randomness, another benefit to use the trusted subsystem for message authentication is that it can use symmetric encryption algorithms such as HMAC in a group, and the group can share the same secret key. This is much more efficient than using an asymmetric digital signature, while still guaranteeing the same non-repudiation feature. This is because the subsystem will neither use other's uid to authenticate its own message, nor disclose its secret key(s) to the untrusted parts of the system, including its host operating system.

3.2.3 Message Certificate and Validation

Every message needs to be proved that it is congruent with the algorithm specification, even if it is correctly authenticated and not equivocating. To achieve this, a process is required to provide a set of previously received messages as a certificate when it broadcasts a new message. Each message in the certificate must have a correct authentication, which has the same non-repudiation feature as a digital signature because of the trusted subsystem. The certificate is piggybacked with the newly sent message. A message is valid only if:

- it can pass the verification of the trusted counter authentication; and

- it includes the correct certificate.

The certificate mechanism is carefully designed to ensure the correctness of the algorithm. An important feature is that the messages included in the certificate do not require further certificate for themselves in a recursive manner. Otherwise, the message size will grow infinitely. This requirement poses a challenge to the certificate design, because a Byzantine process may try to bypass the validity check of a certain message by forging other invalid messages in the certificate, so this attack must be prevented.

The certificate of every message is defined as follows:

1. $\langle INIT, 0, *, v\rangle$ is valid without any certificate.

2. $\langle PR, \phi = 1, *, v, D\text{-}GET\rangle$ requires $\lceil \frac{n-f}{2} \rceil \langle INIT, \phi = 0, *, v\rangle$ for $v = 0$,
   or $\lceil \frac{n-f+1}{2} \rceil \langle INIT, \phi = 0, *, v\rangle$ for $v = 1$.
   Explanation: This is the P-message of the first round after a process has received $n-f$ $INIT$ messages. As shown at line 10, we have a preference over 0 when there is a tie between 0 and 1 among these $n-f$ $INIT$ messages, but this preference can be customized according to the specific application scenarios.

3. $\langle PR, \phi, *, v, D\text{-}GET\rangle (\phi > 1)$ requires $\lceil \frac{n+1}{2} \rceil \langle PR, \phi - 1, *, v, *\rangle$.
   Explanation: This is the P-message after the first round. At first glance, a process i can deterministically get v as long as i has received at least one V-message $\langle VO, \phi - 1, *, v\rangle$ with $v \neq \perp$, as indicated by line 29. However, it is pointless to use only one message as a certificate, because that message may come from a faulty process. The sender i has to answer why that V-message is valid as well. If i is correct, it must have validated that $\langle VO, \phi - 1, *, v\rangle$, whose certificate contains $\lceil \frac{n+1}{2} \rceil \langle PR, \phi - 1, *, v, *\rangle$ as we will see in case 5, so it must include those $\lceil \frac{n+1}{2} \rceil$ messages into its own certificate.

4. $\langle PR, \phi, *, v, R\text{-}GET\rangle (\phi > 1)$ requires $n - f \langle VO, \phi - 1, *, \perp\rangle$.
   Explanation: This can be directly inferred from line 32: only if all $n-f$ processes vote for $\perp$, a process updates its flag to $R\text{-}GET$ and tosses a coin as its proposal in the next round.

5. $\langle VO, \phi, *, v\rangle$ with $v \in \{0, 1\}$ requires $\lceil \frac{n+1}{2} \rceil \langle PR, \phi, *, v, *\rangle$. Furthermore, if there is at least one $\langle PR, \phi, *, v, D\text{-}GET\rangle$ in the certificate, the certificate of this message (see item 2 or 3 above) must also be included.
   Explanation: The message $\langle VO, \phi, i, v\rangle$ implies that the process i has received $\lceil \frac{n+1}{2} \rceil \langle PR, \phi, *, v, *\rangle$ (line 22), which must be included in the certificate. The extra requirement is necessary for the termination, as we will see later in the correctness proof. Intuitively, if i is correct, it must have checked the validity of any $\langle PR, \phi, *, v, D\text{-}GET\rangle$ before putting it into the certificate, so it should strip the certificate of that message and put into its own certificate. This will not lead to recursively adding certificates to the message,

because the certificate of a $\langle PR, \phi, *, v, D\text{-}GET \rangle$ only contains limited number of messages as shown in item 2 and 3. Moreover, if more than one $\langle PR, \phi, *, v, D\text{-}GET \rangle$ messages are included in the certificate of the $\langle VO, \phi, *, v \rangle$, only one of them need to provide its own certificate, because it is enough to prove the validity of all the other $\langle PR, \phi, *, v, D\text{-}GET \rangle$ messages.

6. $\langle VO, 1, *, \perp \rangle$ requires $n - f \langle PR, 1, *, *, D\text{-}GET \rangle$ with both 0 and 1 are proposed and the count of neither value reaches $\lceil \frac{n+1}{2} \rceil$.
   Explanation: This can be directly inferred from line 24, namely a process votes for $\perp$ if it receives $n - f$ proposals with different values. Note that this certificate only applies to round 1. For all further rounds, the case 7 should be applied. This is because only in the first round, all correct processes are guaranteed to deterministically get their proposals. Without the existence of valid *R-GET* proposals, the certificate in this case is simpler than in the following case.

7. $\langle VO, \phi, *, \perp \rangle (\phi > 1)$ requires $n - f \langle PR, \phi, *, * \rangle$ with both 0 and 1 are proposed and the count of neither reaches $\lceil \frac{n+1}{2} \rceil$. It further requires $\lceil \frac{n+1}{2} \rceil \langle VO, \phi - 1, *, \perp \rangle$.
   Explanation: Besides the same certificate as in case 6, the extra $\lceil \frac{n+1}{2} \rceil$ votes for $\perp$ actually constitutes a certificate for $\langle PR, \phi, *, v, R\text{-}GET \rangle$ (see case 4). As we will see later, it is impossible to see both valid $\langle PR, \phi, *, 0, D\text{-}GET \rangle$ and $\langle PR, \phi, *, 1, D\text{-}GET \rangle$ in the same round $\phi > 1$. If a correct process i has received valid proposals of both 0 and 1, at least one of the different values must come from an *R-GET* proposal. Since i is correct, it must have checked the validity of that proposal, whose certificate is $\lceil \frac{n+1}{2} \rceil \langle VO, \phi - 1, *, \perp \rangle$ messages, so i has to put them into its own certificate.

According to the explanation above, we can also conclude the following lemma, which is important to the proof of termination:

Lemma 3 *If a correct process sends a message, it is able to find a corresponding certificate.*

3.3 Correctness proof

In this section we prove the correctness of TRUSTED BEN-OR.

3.3.1 Agreement

We first show that if any correct process decides any value v, then from the next round, no one can generate a valid P-message to propose another value.

Lemma 4 *If a correct process decides v in round ϕ, the only valid P-message of $\phi + k$ is $\langle PR, \phi + k, *, v, D\text{-}GET \rangle$ for any $k > 0$.*

Proof. We prove with induction and start with $k = 1$. A correct process only decides v if there are $\lceil \frac{n+1}{2} \rceil$ valid $\langle VO, \phi, *, v \rangle$. This firstly excludes the existence of any valid $\langle PR, \phi + 1, *, *, R\text{-}GET \rangle$, because such a message requires $\lceil \frac{n+1}{2} \rceil \langle VO, \phi, *, \perp \rangle$ as certificate. Any two sets of $\lceil \frac{n+1}{2} \rceil$ processes must intersect with at least one process, and the trusted counter authentication can prevent any process from voting for both v and $\perp$ in the same round, so the certificate of $\lceil \frac{n+1}{2} \rceil \langle VO, \phi, *, \perp \rangle$ cannot exist. Secondly, a valid $\langle PR, \phi + 1, *, 1 - v, D\text{-}GET \rangle$ cannot exist either. Otherwise, a certificate of $\lceil \frac{n+1}{2} \rceil \langle PR, \phi, *, 1 - v, * \rangle$ must also exist. However, a valid $\langle VO, \phi, *, v \rangle$ contains $\lceil \frac{n+1}{2} \rceil \langle PR, \phi, *, v, * \rangle$ in its certificate. For the same reason of the quorum intersection, at least one process must have proposed both $1 - v$ and v in round ϕ, but this equivocation is impossible because of the trusted counter. Thirdly, $\langle PR, \phi + 1, *, v, D\text{-}GET \rangle$ can be valid because its certificate exists, making it the only valid P-message of $\phi + 1$.

Now assume that the only valid P-message of $\phi + k$ is $\langle PR, \phi + k, *, v, D\text{-}GET \rangle$ for some $k > 0$, then all correct processes only broadcast $\langle PR, \phi + k, *, v, D\text{-}GET \rangle$. This firstly makes $\langle PR, \phi + k + 1, *, 1 - v, D\text{-}GET \rangle$ invalid. Secondly, since all correct processes do not accept any invalid $\langle PR, \phi + k, *, 1 - v, * \rangle$, they will only broadcast $\langle VO, \phi + k, *, v \rangle$. As a result, $\langle PR, \phi + k + 1, *, *, R\text{-}GET \rangle$ cannot be valid because less than $\lceil \frac{n+1}{2} \rceil$ processes vote for $\perp$. Thirdly, $\langle PR, \phi + k + 1, *, v, D\text{-}GET \rangle$ is valid because the $\langle PR, \phi + k, *, v, D\text{-}GET \rangle$ from all correct processes constitutes a certificate. Thus, $\langle PR, \phi + k + 1, *, v, D\text{-}GET \rangle$ is the only valid P-message. Using induction we can confirm this lemma. ∎

The agreement property directly ensues:

Theorem 5 *No two correct processes decide differently.*

Proof. We prove it by contradiction. Suppose that two correct processes decide v in ϕ and $1 - v$ in ϕ' respectively. Apparently $\phi \neq \phi'$, because $\lceil \frac{n+1}{2} \rceil \langle VO, \phi, *, v \rangle$ and $\lceil \frac{n+1}{2} \rceil \langle VO, \phi, *, 1 - v \rangle$ cannot exist at the same time. Assume $\phi < \phi'$, according to Lemma 4, the only valid P-message of ϕ' is $\langle PR, \phi', *, v, D\text{-}GET \rangle$. However, if a correct process decides $1 - v$ in ϕ', it must have received $\lceil \frac{n+1}{2} \rceil$ valid $\langle VO, \phi', i, 1 - v \rangle$ messages certified with $\lceil \frac{n+1}{2} \rceil \langle PR, \phi', *, 1 - v, * \rangle$. This leads to a contradiction, because no correct process will propose $1 - v$ in a P-message. ∎

3.3.2 Termination

The proof of termination is inspired by Aguilera and Toueg [3] who have proved the original Ben-Or's algorithm, but our proof is more complex due to the Byzantine behaviors. We first show that every correct process is able to start any round, then prove that there is eventually a "lucky" round in which every correct process can decide.

Lemma 6 *Every correct process is able to start any round $\phi \geq 0$.*

Proof. It clearly applies for $\phi = 0$, namely the $INIT$-round. After all the $INIT$-messages from at least $n-f$ correct processes arrive, any correct processes can finish the wait of line 9 and then start $\phi = 1$. Now assume that every correct process starts a round $\phi \geq 1$. According to Lemma 3, each correct process is able to assemble a certificate and broadcast a valid P-message. So eventually there are at least $n-f$ valid P-messages in the system, enabling correct processes to terminate the wait of line 21 and then broadcast a valid V-message. Again there are at least $n-f$ valid V-messages eventually, so every correct process can terminate the wait of line 26 and start the next round $\phi + 1$. Using induction we confirm that every correct process can start any round $\phi \geq 0$. ∎

Corollary 7 *In every round ϕ, at least one of the three P-message forms is valid: $\langle PR, \phi, *, 0, D\text{-}GET \rangle$, $\langle PR, \phi, *, 1, D\text{-}GET \rangle$, $\langle PR, \phi, *, *, R\text{-}GET \rangle$. And at least one of the three V-message forms is valid: $\langle VO, \phi, *, 0 \rangle$, $\langle VO, \phi, *, 1 \rangle$, $\langle VO, \phi, *, \perp \rangle$*

In order to show a lucky round will eventually happen, we adopt a similar definition used in the proof of the original Ben-Or's algorithm [3], but with some modifications due to the presence of the Byzantine faults:

Definition 7 *A value $v \in \{0,1\}$ is ϕ-**major** at time t_0, if $\geq \lceil \frac{n+1}{2} \rceil$ processes have created the message $\langle PR, \phi, *, v, * \rangle$ and authenticated with the monotonic counter at t_0. A value $v \in \{0,1\}$ is ϕ-**locked** at time t_0, if 1) no valid $\langle PR, \phi, *, 1-v, * \rangle$ with a certificate exists before t_0 and 2) we can prove that from t_0 on, no $\langle PR, \phi, *, 1-v, * \rangle$ can be created, or such a message can never collect a certificate.*

Note that whether a value is ϕ-major or not can be directly determined by the current system status, namely by counting all the properly authenticated P-messages of round ϕ. A ϕ-locked value, however, needs to be proved because we have to take the status in the future into account. In other words, v is ϕ-locked at t_0 means that we are sure that no valid $\langle PR, \phi, *, 1-v, * \rangle$ can exist at all. Obviously, if v is ϕ-major or ϕ-locked at t_0, then v is also ϕ-major or ϕ-locked at any time $t \geq t_0$.

Lemma 8 *If a value v is ϕ-locked at some time, then every correct process can decide v by the end of round $\phi + 1$.*

Proof. All correct processes will start round ϕ and they only propose v. They will not accept any P-message with value $1-v$ in round ϕ, since it is invalid. Now that they only accept P-messages with value v, they only vote for v in round ϕ, leading to less than $\lceil \frac{n+1}{2} \rceil \langle VO, \phi, *, \perp \rangle$. Therefore for the next round, $\langle VO, \phi + 1, *, \perp \rangle$ can never become valid. Neither $\langle VO, \phi + 1, *, 1-v \rangle$ can be valid, because of the lack of certificate. According to Lemma 6 and Corollary 7, a correct process can complete collecting valid V-messages of round $\phi + 1$ at line 26, and the only valid form is $\langle VO, \phi + 1, *, v \rangle$. As a result, every correct process can decide v by the end of $\phi + 1$. ∎

In the remaining part we will show that if the trusted random number generators of every process happen to generate a sequence of lucky results, one value will become locked. We start with the following lemma:

Lemma 9 *If v is ϕ-major at t_0, and if the trusted random number generator happens to output v for every process creating $\langle PR, \phi+1, *, v, R\text{-}GET\rangle$ at any time (can be before t_0), then v is $(\phi+1)$-locked at t_0.*

Proof. There is no $\langle PR, \phi+1, *, 1-v, R\text{-}GET\rangle$ because of the trusted random number generator. And a $\langle PR, \phi+1, *, 1-v, D\text{-}GET\rangle$ cannot be valid any more, because v is already ϕ-major and no certificate containing $\lceil \frac{n+1}{2} \rceil \langle PR, \phi, *, 1-v, *\rangle$ can exist. ∎

Starting from $\phi = 2$, we group every 3 rounds into an *epoch*. For example, the r-th epoch ($r \geq 1$) consists of rounds $3r-1$, $3r$ and $3r+1$. Now we define two oracle functions for the purpose of the proof. The oracles can query the state of the whole system when we invoke them, but are not available to the processes.

The `first_toss`(r, t) oracle returns the time $t_a \leq t$, when the first correct process executes line 20 to flip a coin to create $\langle PR, 3r, *, v, R\text{-}GET\rangle$ in round $3r$. If no correct processes ever did that, the oracle returns NaN. Note that the correct process i needs only to be correct until t_a. The function can return $t_a = t$, i. e. a correct process is executing line 20 exactly at time t.

The `lucky_coin`$(r, \phi, t) \rightarrow \{0, 1\}$ oracle assesses whether a random bit obtained in round ϕ, at time t is lucky or not. The return value of `lucky_coin`(r, ϕ, t) is defined as following:

Definition 8 *(i)* `lucky_coin`$(r, 3r+1, t)$ *returns 1 for any t;*

(ii) `lucky_coin`$(r, 3r-1, t)$ *and* `lucky_coin`$(r, 3r, t)$ *return 1, if* `first_toss`(r, t) *returns t_a, and 0 is not $(3r-1)$-major at time t_a;*

(iii) `lucky_coin`$(r, 3r-1, t)$ *and* `lucky_coin`$(r, 3r, t)$ *return 0 in cases other than (ii), i. e. either* `first_toss`(r, t) *returns NaN, or* `first_toss`(r, t) *returns t_a and 0 is already $(3r-1)$-major at time t_a.*

Remark 3 *Obviously, as soon as a process (correct or Byzantine) has executed line 20 to flip the coin, we can immediately know whether the result is lucky or not. The reason is that the return values of both* `first_toss`(r, t) *and* `lucky_coin`(r, ϕ, t) *are determined merely by the events happened before or at t, and are independent of any future events. As we will discuss later, this property is crucial to correctly prove the termination.*

Definition 9 *An epoch is **lucky**, if every coin toss of line 20 in this epoch at any time t gets the consistent result of the* `lucky_coin`(r, ϕ, t).

Let t_b be the time when every correct process has completed round $3r+1$. Such a t_b must exist (Lemma 6), so we can categorize the system state according to the time of the first correct coin toss of round $3r$ before or at t_b. There are only three possibilities:

1. `first_toss`(r, t_b) returns NaN, meaning that no correct process ever tossed a coin in round $3r$;
2. `first_toss`(r, t_b) returns t_a and 0 is $(3r-1)$-major at time t_a;
3. `first_toss`(r, t_b) returns t_a and 0 is not $(3r-1)$-major at time t_a.

For the three cases, there are the following lemmas:

Lemma 10 *(Case 1) In a lucky epoch r, if* `first_toss`(r, t_b) *returns NaN, then some value v is $(3r+1)$-locked at t_b.*

Proof. All correct processes have completed phase $3r$ and no one created $\langle PR, 3r, *, *, R\text{-}GET\rangle$, so they all must have created $\langle PR, 3r, *, v, D\text{-}GET\rangle$ with the same v. As a result, `first_toss`(r,t) should always return NaN for any t, so `lucky_coin`$(r, 3r, t)$ must always return 0 due to Definition 8 (iii). Now we show that the Byzantine processes cannot prevent v becoming $(3r+1)$-locked.

If $v = 0$: Firstly, there are no valid $\langle PR, 3r, *, 1, D\text{-}GET\rangle$ messages, because $\langle PR, 3r, *, 0, D\text{-}GET\rangle$ from the correct processes must be valid. Secondly, if a Byzantine process creates a $\langle PR, 3r, *, v', R\text{-}GET\rangle$ messages, there must be $v' = 0$ because of the lucky coin. Therefore, 0 is $3r$-locked, thus also $(3r+1)$-locked.

If $v = 1$: Since all correct processes propose 1 in round $3r$, 1 is $3r$-major at time t_b. According to case (i) of the definition of `lucky_coin`, in round $3r+1$, the lucky coin always returns 1. Thus, 1 is $(3r+1)$-locked because of Lemma 9. ∎

Lemma 11 *(Case 2) In a lucky epoch r, if* `first_toss`(r, t_b) *returns t_a and 0 is $(3r-1)$-major at time t_a, then 0 is $3r$-locked at t_b.*

Proof. For any time $t < t_a$, `first_toss`(r,t) should return NaN; for any time $t \geq t_a$, `first_toss`(r,t) should return t_a. According to Definition 8 (iii), `lucky_coin`$(r, 3r, t)$ must return 0 for any t. That means, every $\langle PR, 3r, *, v, R\text{-}GET\rangle$, whenever it is created, must have $v = 0$. According to Lemma 9, 0 must be $3r$-locked. ∎

Lemma 12 *(Case 3) In a lucky epoch r, if* `first_toss`(r, t_b) *returns t_a and 0 is not $(3r-1)$-major at time t_a, then 1 is $(3r+1)$-locked at t_b.*

Proof. A correct process tosses the coin and creates a $\langle PR, 3r, *, v, R\text{-}GET\rangle$ at time t_a. Here $v = 1$ is the lucky value according to Definition 8 (ii). This indicates that it must have received

$\lceil \frac{n+1}{2} \rceil \langle VO, 3r-1, *, \perp \rangle$, among which at least one is from a correct process. That process must have received at least one valid $\langle PR, 3r-1, *, 1, * \rangle$. This valid P-message cannot have the flag *R-GET*, because any lucky coin tossed before t_a must get 0 (Definition 8 (iii)), so it must be a $\langle PR, 3r-1, *, 1, D\text{-}GET \rangle$. This can exclude any valid $\langle PR, 3r-1, *, 0, D\text{-}GET \rangle$ forever.

Now we prove that in round $3r$, every correct process will only create $\langle PR, 3r, *, 1, * \rangle$. Assume, for the sake of contradiction, a correct process can create $\langle PR, 3r, *, 0, * \rangle$ at some time t. In a lucky epoch, this message cannot have the flag *R-GET*, because no correct process tosses its coin in round $3r$ before t_a, and from t_a on, `lucky_coin`$(r, 3r, t)$ always returns 1. Therefore, it must be a $\langle PR, 3r, *, 0, D\text{-}GET \rangle$. According to the algorithm specification, this correct process must have received at least one valid $\langle VO, 3r-1, *, 0 \rangle$. Recall the definition of the certificate: a valid $\langle VO, 3r-1, *, 0 \rangle$ requires $\lceil \frac{n+1}{2} \rceil \langle PR, 3r-1, *, 0, * \rangle$ as a certificate. Furthermore, if there is at least one $\langle PR, 3r-1, *, 0, D\text{-}GET \rangle$ in the certificate, the certificate of that message must also be included. Consider the following two sub-cases with respect to the certificate of the valid $\langle VO, 3r-1, *, 0 \rangle$:

Sub-case 1: all the $\lceil \frac{n+1}{2} \rceil \langle PR, 3r-1, *, 0, * \rangle$ have the flag *R-GET*. This is impossible, because at time t_a, 0 is not yet $(3r-1)$-major, whilst after t_a, the lucky coin for round $3r-1$ always returns 1. Consequently, the number of $\langle PR, 3r-1, *, 0, R\text{-}GET \rangle$ messages can never reach $\lceil \frac{n+1}{2} \rceil$.

Sub-case 2: there is at least one $\langle PR, 3r-1, *, 0, D\text{-}GET \rangle$ among the $\lceil \frac{n+1}{2} \rceil \langle PR, 3r-1, *, 0, * \rangle$ certificate, then this *D-GET* message itself must also be certified. However, as we have seen at the beginning, any $\langle PR, 3r-1, *, 0, D\text{-}GET \rangle$ message cannot be valid, which leads to a contradiction.

We can conclude that none of the correct processes will create a $\langle PR, 3r, *, 0, * \rangle$, neither will they got stuck, so all of them will eventually create $\langle PR, 3r, *, 1, * \rangle$ messages, making value 1 to be $3r$-major. Considering that the lucky coin always returns 1 in round $3r+1$, 1 must be $(3r+1)$-locked because of Lemma 9. ∎

Lemma 13 *If an epoch r is lucky, all correct processes can decide no later than round $3r+2$.*

Proof. According to Lemmas 10, 11 and 12, some value v must be $3r$-locked or $(3r+1)$-locked at some time, so all correct processes can decide in round $3r+1$ or $3r+2$ (Lemma 8). ∎

Theorem 14 *The probability that all correct processes decide is 1.*

Proof. An epoch is lucky if and only if all results from the random number generator coincide with the definition of the `lucky_coin` oracle. Each of these coincidences has the probability of 0.5, and is independent with each other. Every epoch may contain at most $3n$ random numbers, so the probability that an epoch is lucky is at least $(0.5)^{3n}$, so the probability that a lucky epoch eventually occurs is $1-(1-(0.5)^{3n})^{\infty} = 1$. Lemma 13 ensures that all correct processes must decide immediately after such a lucky epoch. ∎

3.3.3 Validity

Theorem 15 *if a correct process decides v, v must be proposed by at least $\lceil \frac{n-f}{2} \rceil$ processes.*

Proof. Assume v is proposed by less than $\lceil \frac{n-f}{2} \rceil$ processes, then there is no valid $\langle PR, 1, *, v, * \rangle$ in the first round, which means v is 1-locked. Every correct process will decide $1-v$ by the end of round 2 (Lemma 8), so no correct one will decide v. ∎

3.4 Discussion and Optimization

Now we discuss some common issues in the proof of randomized consensus algorithms, as well as some optimization strategies.

3.4.1 When Randomization Meets a Strong Adversary

Like most randomized and round-based algorithms, the termination of TRUSTED BEN-OR relies on a set of processes to luckily obtain the preferred coin values in certain rounds. The definition of a lucky coin value in Definition 8 is not trivial, but we argue that this is a correct one in a strong adversary model. As mentioned in Remark 3, the luckiness of a coin only depends on the current system state, but not on any future events. Otherwise, suppose that a coin is only lucky if something in the future happens, the adversary could take actions to prevent this from occurring.

To address this issue, we give another idea to prove the termination of TRUSTED BEN-OR. The termination can be ensured by the following facts:

- If all valid proposals have the same value in some round, the processes can later decide (Lemma 8).
- If any two valid proposals in some round (>1) have their values deterministically obtained (D-get), they must have the same value (the certificate mechanism).
- With certain probability, there is a lucky round where all random proposals (R-get values) happen to have the same value v, which coincides with the value of the valid deterministic proposals (if there are any), then all valid proposal have the same value v in this round. Consequently, the processes can later decide.

This proof sketch looks much simpler. The similar idea can also be found in the proof of the Turquois algorithm [51] as well as in textbooks to prove Ben-Or's algorithm [13, pp.238–242]. At first glance, it looks alright. However, the third statement implies that if some process deterministically gets a value, this must happen before another process tosses the coin. If this timing assumption does not hold and some process firstly tosses a coin, we cannot know whether it is lucky or not, because it depends on an event in the future. Thus, we have to argue why this assumption can always hold, especially when a strong adversary exists.

We do not assert that all algorithms using the above-mentioned proof technique are problematic under a strong adversary model, but unfortunately, there are real examples of failed cases. One of the examples is a variant of Ben-Or's algorithm using a global coin [3]. In this variant, all processes have access to a shared coin in each round, instead of tossing their own coins independently. This optimization is believed to be able to accelerate the termination, because the probability that all those processes tossing the coin get the same lucky value is higher. However, using this global coin can lead to non-termination if there is a strong adversary. The readers are encouraged to refer to the formal proof [3]. The idea is that if the processes initially have different values, the adversary can firstly let a process randomly obtain a value v by manipulating the message delivery order, then let another process deterministically update its value to $1-v$ in the same round. For the remaining processes, the adversary can later freely choose to let them get v via the global coin, or get $1-v$ deterministically. In the next round, the adversary can again manipulate the deterministic value after seeing the result of the global coin of that round. Consequently, the processes enter a new round with different values again, and the adversary can repeatedly play this trick forever. However, if we adopt the proof sketch mentioned above, we can still prove its termination. The problem lies in the fact that when a process tosses a coin before any deterministic value occurs in a round, we cannot tell whether this result is lucky or not. In Definition 8 used in our proof, we get rid of this issue. As stated in Remark 3, the luckiness can be immediately checked under the current system state and does not rely on any future events.

3.4.2 Handling Omission Failures

In real-world networks, especially in wireless ad hoc networks, links are not always reliable and messages could get lost. For example, we can consider a *fair-loss link* model [13, pp.34–35], in which the communication link between two processes can drop any subset (but not all) of the messages transferred via the link. More precisely, it requires that if a process p sends a message infinitely many times to q, q will then deliver the message infinitely many times. TRUSTED BEN-OR cannot work under a fair-loss link model, as we can consider the following example. All processes are executing without any message omission, and suddenly process p is separated from the others for a while and missed all messages in between. During this network partition, the other processes can continue working and have entered a more advanced phase than p. As a result, p is left behind and gets stuck, even if it is connected to the others later, because the messages it needs to make a progress are lost. This issue can be solved by helping a process who is left behind to catch up with others, so we modify the algorithm by introducing another two tasks running in parallel to Algorithm 3.2:

- Task 1 is to periodically broadcast the message that the process has last sent.
- Task 2 is to "jump" to a future round or phase, if it has received a valid message from that round or phase.

More specifically, if a process receives a valid $\langle PR, \phi', *, v, D\text{-}GET\rangle$ or $\langle VO, \phi', *, v\rangle$ that is more advanced than its current state, it will use the trusted subsystem to authenticate a message with the same content, then broadcast the message and update its state correspondingly. If a valid $\langle PR, \phi', *, *, R\text{-}GET\rangle$ is received, the process has to invoke the `authenticate_with_coin`$(\langle PR, \phi', p, \square, R\text{-}GET\rangle, [\phi'|0])$ to toss its own trusted coin, and then broadcast the message and update its state. The received message must be valid: correctly authenticated by the trusted subsystem and with a certificate. The certificate is used to certify the newly generated message and sent alongside the message. This can prevent a Byzantine process from forging a message out of thin air to bring a correct process to an inconsistent state.

With this modification, the agreement of the algorithm still holds, because the correctness of Lemma 4 and Theorem 5 only relies on the equivocation prevention and certificate mechanism. However, the termination becomes problematic. One thing we can guarantee is that no correct process gets blocked at any round:

Lemma 16 *Assume the processes are connected via fair-loss links. At any time t_1, for any correct process in round ϕ_1, there is a time $t_2 > t_1$ so that the process enters a new round $\phi_2 > \phi_1$ at t_2.*

Proof. To prove by contradiction, assume there is a correct process p that stays forever in round ϕ_1 after time t_1. Then apparently there is no correct process entering any round $\phi_2 > \phi_1$, otherwise that process will periodically broadcast messages of round ϕ_2 or later rounds. Under the fair-loss link model, eventually p can receive at least one of them and can jump to a new round, which violates our assumption. That means, all other correct processes keep staying in rounds $\leq \phi_1$ after time t_1. Because p is infinitely broadcasting its message of ϕ_1, all correct processes will eventually receive it and will enter and stay in ϕ_1. Eventually at least one correct process can receive the messages from all correct processes, and can then enter $\phi_1 + 1$. This leads to a contradiction. ∎

However, this is not enough for termination. Consider such a scenario where all processes except for p are working correctly and without any message omission. As for p, it receives all P-messages but misses all V-messages from others in each round. This does not violate the characteristics of the fair-loss link, because p *does* receive infinite messages (all the P-messages) among the infinite number of messages sent by other processes (both P-messages and V-messages). However, p can never decide.

The good news is that in the real world, we can hardly encounter the corner case mentioned above. Compared to fair-loss link, a more practical assumption is that every single message can get lost with a certain probability. The similar assumption is adopted in [69]. In this case, if there is a lucky epoch and if all messages in the epoch plus the next round are all successfully delivered, then all the correct processes can decide. As we will see later, during our experiments we have also observed regular packet losses (24%) in a real-world network, but our algorithm with the modification can still terminate with relatively low latency.

3.4.3 Decision Forwarding

The original algorithm requires that each process keep running even it has decided, because others may need its help to decide. According to Lemma 4 and Lemma 8, if any correct process decides in round ϕ, all correct processes can decide no later than the round $\phi + 1$. However, this guarantee for termination cannot hold anymore if we take message omission into account. A process can only decide after it has received $\lceil \frac{n+1}{2} \rceil$ valid V-messages voting for the same value. Because each message can probabilistically get lost, the chance that a non-decided process successfully receives $\lceil \frac{n+1}{2} \rceil$ such messages in the same round can be small. To accelerate the termination and help others to decide more quickly, a decided process p can repeatedly broadcast its decision $\langle DE, p, v \rangle$. The decision does not require a specific counter authentication, but has to include the quorum of vote messages as a certificate, namely $\lceil \frac{n+1}{2} \rceil \langle VO, \phi, *, v \rangle$ from the same round ϕ and with the same value v. The V-messages in the certificate do not require further certificates, because at least one is from a correct process. This modification does not break the agreement, because the proof of Lemma 4 relies on the condition that $\lceil \frac{n+1}{2} \rceil \langle VO, \phi, *, v \rangle$ exist and at least one among them is valid. From this condition, it can be ensured that no two correct processes decide differently. As for the termination and validity, the correctness is trivial. With this decision forwarding mechanism, a non-decided process can immediately decide upon receiving a single valid $\langle DE, *, v \rangle$, and does not need to wait for a quorum of V-messages from different senders.

3.5 Evaluation

In this section, we present the evaluation results of TRUSTED BEN-OR.

3.5.1 Testbed and Methodology

We build a testbed consisting of ten nodes of Raspberry Pi 3 (model B), connected in a wireless ad-hoc network. The hardware specification of each node is listed below:

- Quad Core ARM Cortex-A53 CPU (1.2GHz clock)
- 1 GB LPDDR2 RAM
- 2.4 GHz 802.11n wireless module

The trusted subsystem is built on top of the Open Portable Trusted Execution Environment (OP-TEE) [67] based on ARM TrustZone [5]. [1] OP-TEE is an ARM-architecture-based TEE that is owned by the TrustedFirmware.org project (by the year 2020 when this thesis is written). It provides a Linux kernel TEE framework and necessary drivers and libraries that run in the secure

[1] Strictly speaking the Raspberry Pi is not trustworthy enough. It only provides ARM TrustZone exception status, but lacks the hardware support for secure boot and memory protection. We only use this hardware for demonstration purposes.

Table 3.1: Latency of authenticate/verify in ms

Message size	128 B	512 B	1024 B	4096 B
Authenticate	0.398	0.406	0.419	0.619
Verify	0.352	0.376	0.410	0.594

world and in parallel to the non-secure operating system in the normal world. It implements the APIs that defines how a user from the normal world can communicate with the trusted application in the secure world. For more details about the OP-TEE, the readers can refer to the its documentation [67].

The implementation is divided into two parts. The TRUSTED BEN-OR algorithm described in Algorithm 3.1 is implemented in C++ — except for the trusted functions, i. e. the counter authentication and random number generation. The network communication I/O is implemented with the Boost.ASIO library (version 1.65.0) [8].

The trusted subsystem containing the counter authenticator is implemented in C, which is the only supported language for programming in the secure world in OP-TEE (at least at the time we conduct the experiment). The trusted counter authenticator uses the SHA-256-based HMAC algorithm for authentication. The cryptographic functions are already included in OP-TEE. The secret key is correctly distributed before the experiments start. All the nodes share the same key, because even a malicious host cannot access the secret key protected by the trusted subsystem.

When the TRUSTED BEN-OR algorithm wants to invoke the trusted counter authenticator from the normal world, the program must execute a special *SMC* instruction to issue an exception. The exception is then trapped by the monitor of the secure world. The monitor then switches to the secure world to execute the trusted function. After the called trusted function returns, the execution will switch back to the normal world. All the above-mentioned procedures are managed by OP-TEE. Because the normal world has no direct access to the memory region in the secure world, shared memory is used to pass parameters and results between the two worlds. The caller from the normal world specifies the pointers to the shared memory chunk for storing the parameters. Besides, the caller can also specify a pointer to another chunk to receive the results.

After the trusted subsystem is implemented, we firstly test the performance of the trusted counter authenticator. The average latency of message authentication and verification is listed in Table 3.1. For messages up to 4 KB, the latency of both authentication and verification is about 0.6 ms, which is negligible compared to the communication delay (shown later).

As a comparison, we implement Turquois in C++ on the same system as well, but exclusively in the normal world because it does not rely on the trusted subsystem.

The ten nodes are distributed in different rooms in our office building and their farthest distance is about 20 meters. They are connected with a wireless ad hoc network, and all messages of TRUSTED BEN-OR are sent via UDP multicast. The minimal, median and maximum of the round trip time of an ICMP ping message is 5.6 ms, 12.5 ms and 1356.7 ms respectively. With

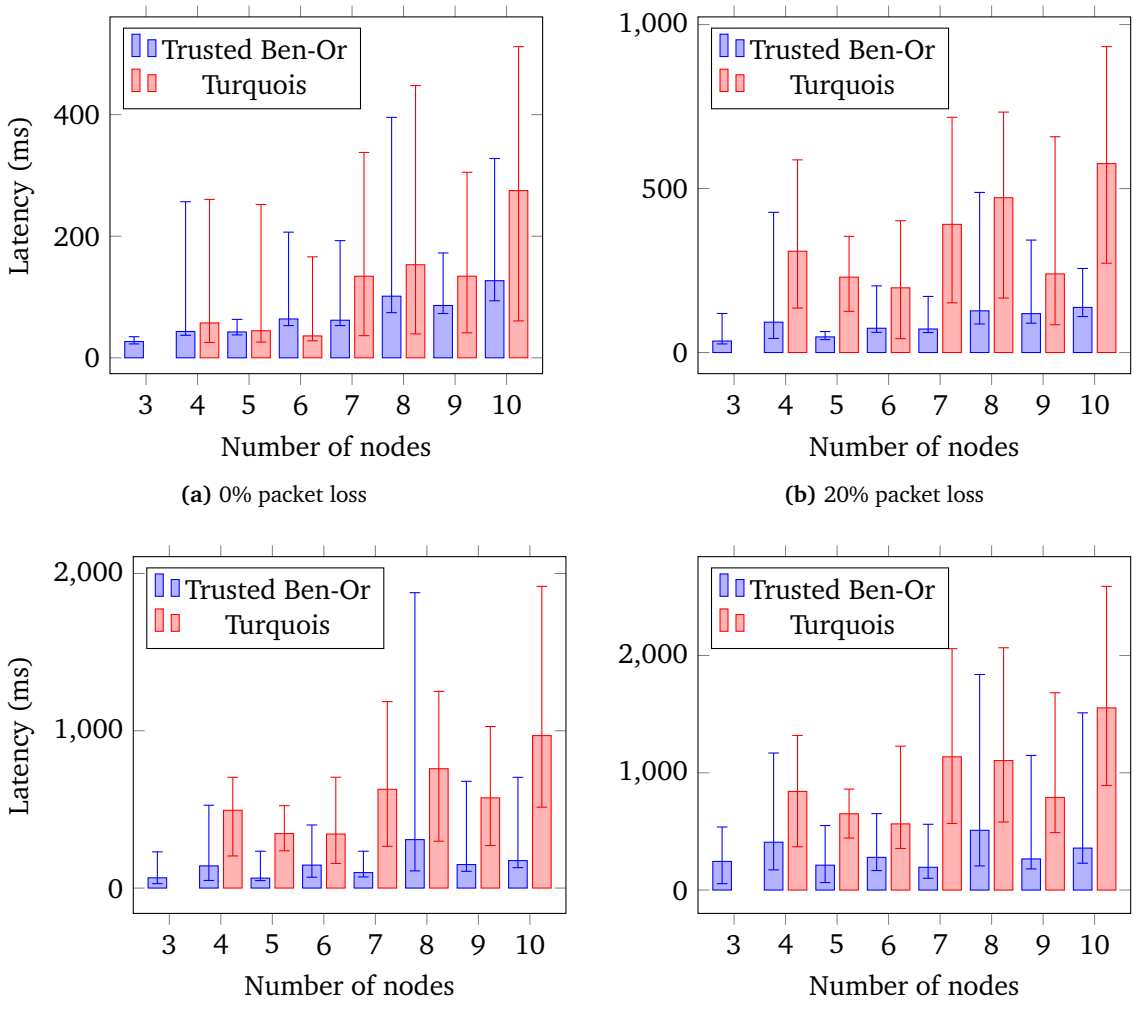
the *iperf3* tool we also test the UDP link, and the result between two nodes reports the jitter as 139.9 ms, which stands for a high variance of the communication delay, and 24% packet loss rate. So compared to a simulated network, this testbed can reflect a real network environment more closely. Because of the packet loss, we implemented the optimization of Section 3.4.2. Further more, up to 60% extra packet losses are introduced in the experiment, to test the resilience of TRUSTED BEN-OR against severe network condition. The packet loss is simulated at the receiver's side, namely after receiving every message, a node may immediately drop it with the specified probability.

For the experiment, all nodes are connected to a signal machine via Ethernet, and wait for the start signal to start the consensus algorithm almost simultaneously. The nodes are equally divided to be assigned with 0 and 1 as their initial proposals. The performance is evaluated by the termination latency, i. e. the duration from the time when the nodes start the algorithm until *all* non-faulty nodes decide. For each system setting, we repeat the experiment 100 times.

3.5.2 Experiment with Non-Faulty Processes and Omission Faults

Firstly the fault-free case is evaluated, and the results are compared to Turquois in Figure 3.1. There is no faulty process, but we inject 0%, 20%, 40% and 60% extra packet losses in the test cases. The bar of Turquois only starts with 4 as it cannot work with 3 nodes. The error bar represents the 9th and 91st percentiles. A big variance and some outliers can be observed in the results, because it is in a real-world environment. Many factors, especially the network condition, can influence the results, although we repeated each test case many times. The results show that in the minimum group with only three nodes and without extra packet loss, the algorithm can terminate in only 26 ms, which is roughly twice the average message round-trip-time. As long as the network condition is not too severe, namely with $\leq 40\%$ packet loss, the median of latency is within 200 ms, except for the outlier of 8 nodes and 40% packet loss. Furthermore, TRUSTED BEN-OR outperforms Turquois in almost every test case, especially when the group size becomes bigger. For example in a group of 10 nodes, the median latency of TRUSTED BEN-OR is about one half of that of Turquois (126 ms vs. 275 ms) with 0% packet loss, while with 60% packet loss, the ratio decreases to 0.22 (358 ms vs. 1553 ms). The reason is TRUSTED BEN-OR only needs two phases in each voting round, instead of three phases in Turquois, and the quorum size is of TRUSTED BEN-OR is smaller.

To clearly show how the performance is impacted by the group size and network condition, we gather the median latency of TRUSTED BEN-OR in all test cases in Figure 3.2. The result is not surprising. Namely, when the group size grows, and the packet loss rate increases, the latency tends to grow as well. Surprisingly, there is an obvious fluctuation of the latency as the group size increases one by one. More precisely, the latency will drop when the group size increases from an even number to an odd number. This phenomenon can be explained by the quorum size. For both group sizes $2n$ and $2n+1$, the quorum size is $n+1$, which is the majority, but in the latter



(a) 0% packet loss

(b) 20% packet loss

(c) 40% packet loss

(d) 60% packet loss

Figure 3.1: Median of latency of TRUSTED BEN-OR in fault-free case in comparison to Turquois. The error bar represents the 9th and 91st percentile.

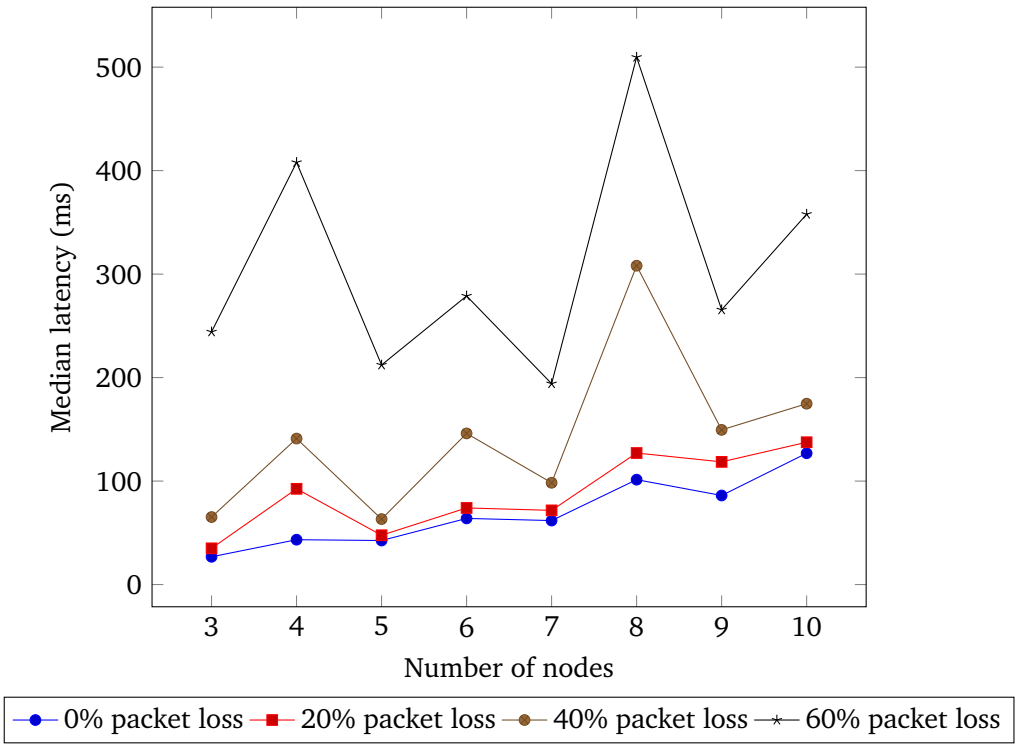
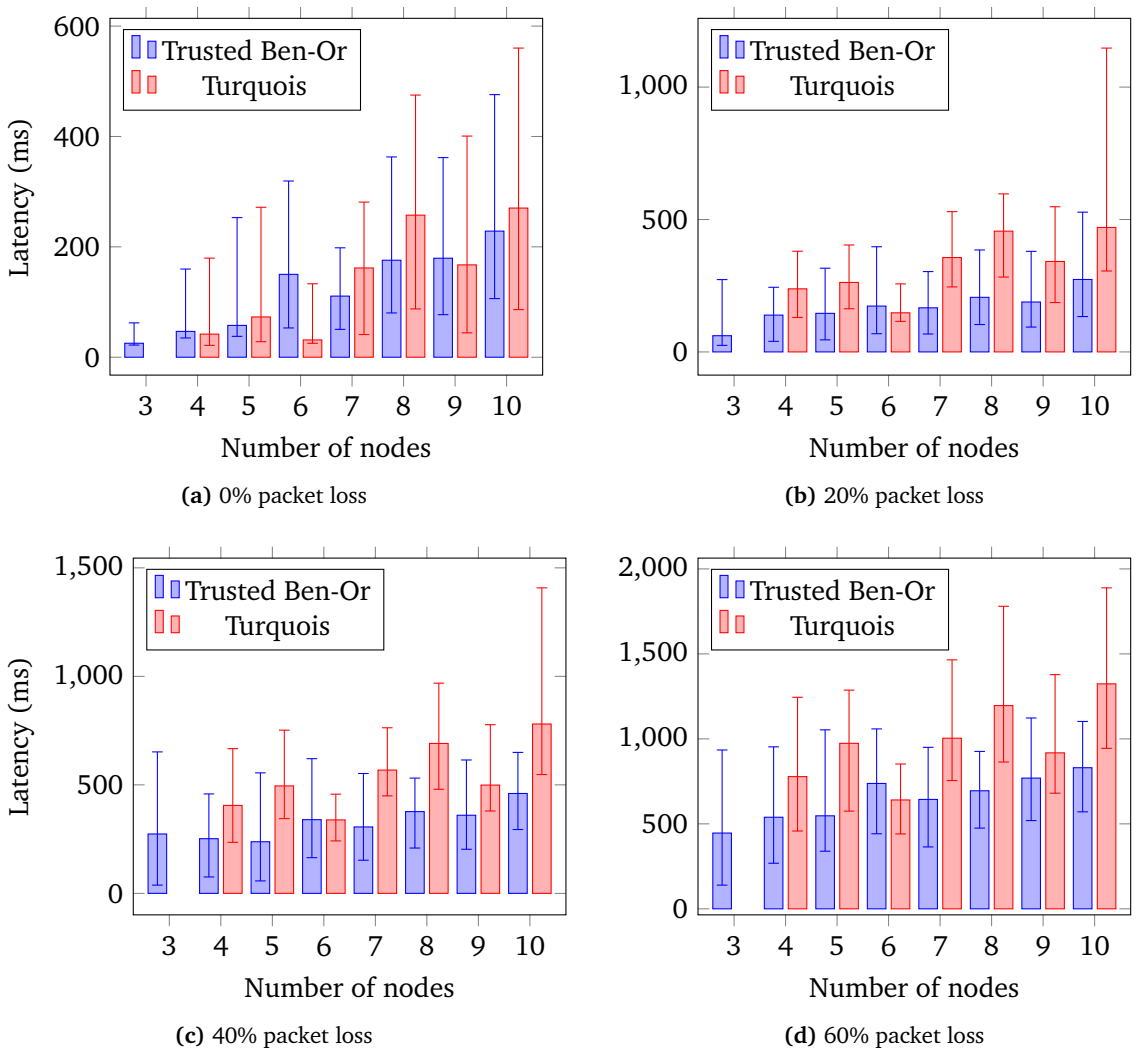


Figure 3.2: Median of latency of TRUSTED BEN-OR in fault-free case.

case there is one more "message provider" to accelerate the termination, especially when the messages can get lost. As a result, the nodes in the latter case can more easily decide. A detailed explanation will be given in the following subsection.

3.5.3 Experiment with Byzantine Processes and Omission Faults

In the second experiment, we inject Byzantine faults into the system to evaluate the fault resilience of TRUSTED BEN-OR. More specifically, we let $\lfloor \frac{n-1}{2} \rfloor$ nodes act as Byzantine processes. Whenever they are about to send a value of 0 or 1, they flip the value to the opposite and then send it; and if they are sending a $\perp$, they do not change it. Note that if the value from a trusted coin is flipped, a correct node will notice this because the verification of the authentication code will fail. Furthermore, we let Byzantine nodes not perform the validation at all, so that they can collude with each other by including invalid messages in the certificate, especially when they are about to vote for $\perp$. As a result, there are less valid messages with value 0 or 1, while more valid messages voting for $\perp$. This will hinder the correct nodes achieving consensus. We inject faults into Turquois as well, but only $\lfloor \frac{n-1}{3} \rfloor$ nodes are Byzantine. Since there is no equivocation prevention mechanism in Turquois, we let faulty nodes always send two opposite values in the same message rounds, so that different recipients may receive different values.



(a) 0% packet loss

(b) 20% packet loss

(c) 40% packet loss

(d) 60% packet loss

Figure 3.3: Median of latency of TRUSTED BEN-OR with Byzantine nodes in comparison to Turquois. The error bar represents the 9th and 91st percentile.

We compare the two algorithms with the existence of Byzantine faults and the results are shown in Figure 3.3. Still, TRUSTED BEN-OR uses less time to achieve consensus, but the difference is not as much as in the fault-free cases. This is because we always set the maximum number of Byzantine nodes, so in general, there are more Byzantine nodes in TRUSTED BEN-OR than in Turquois with the same group size.

Figure 3.4 shows the latencies of all different group sizes and packet loss rates respectively. We can observe that the fluctuation between an odd and even group size is less obvious in this experiment compared to that in a fault-free case. Recall that in a fault-free case, the latency decreases from group size $2n$ to $2n+1$, because the quorum size in both groups keeps the same, but in the latter group there is one more "message provider". In a Byzantine case however, this extra message provider is actually a Byzantine node, which cannot contribute to termination.

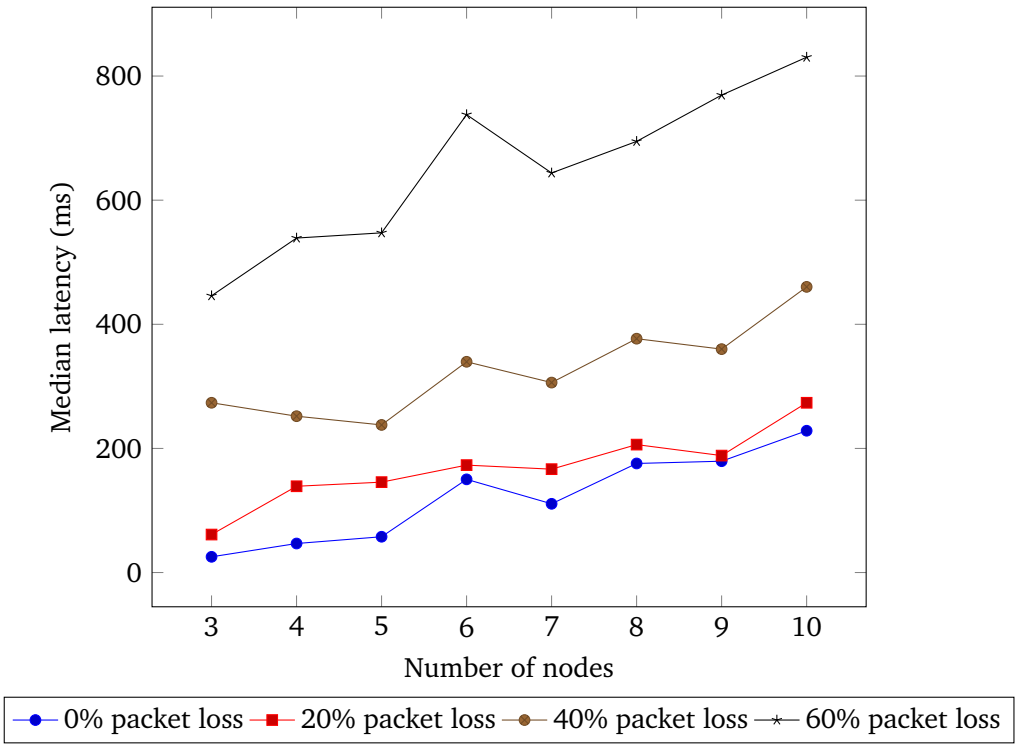


Figure 3.4: Median of latency of TRUSTED BEN-OR with Byzantine nodes.

The reason is that the number of faulty nodes is always set to the maximum, so in a group of $2n$ nodes, we put $n-1$ faulty ones there while in a group of $2n+1$, there are n faulty nodes.

We can conclude that by utilizing a trusted subsystem, TRUSTED BEN-OR is more efficient and at the same time can tolerate more faulty processes than Turquois in most cases. There are mainly two reasons. Firstly, because of the trusted subsystem, Byzantine processes cannot behave in an arbitrarily faulty manner, especially they cannot equivocate. Thus, more faulty processes can be tolerated, and it also makes the quorum smaller and makes the message validation much simpler. Secondly, TRUSTED BEN-OR only needs two phases in each voting round, compared to three phases in Turquois. This leads to less communication cost.

3.5.4 An Explanation About the Difference Between Odd and Even Group Sizes

If we take the potential communication failures into account, a bigger quorum size indicates that it is normally harder to collect messages from a quorum. To show this, we can consider a simplified situation as following:

1. All processes are correct and each sends a valid message;
2. Each message may get lost with the same probability p;
3. Each process is waiting for at least $\lceil \frac{n+1}{2} \rceil$ messages.

We consider two group sizes $n = 2k$ and $n = 2k+1$ for the same $k \geq 2$, and compare the probability that the recipient can successfully deliver at least $\lceil \frac{n+1}{2} \rceil$ messages. The quorum size $\lceil \frac{n+1}{2} \rceil = k+1$ is the same for both group sizes. We use a random variable X to denote the number of messages that are delivered to the certain recipient, so $Pr(X \geq k+1)$ is equivalent to the probability that this recipient successfully collects a quorum of messages.

The probability that *exactly* m messages are delivered is calculated as $Pr(X = m) = \binom{n}{m}(1-p)^m p^{n-m}$, so with $n = 2k$ and $n = 2k+1$, the probabilities that at least $k+1$ messages are delivered can be calculated as:

$$Pr(X \geq k+1 \mid n = 2k) = \sum_{m=k+1}^{2k} \binom{2k}{m}(1-p)^m p^{2k-m} \tag{3.1}$$

$$Pr(X \geq k+1 \mid n = 2k+1) = \sum_{m=k+1}^{2k+1} \binom{2k+1}{m}(1-p)^m p^{2k+1-m} \tag{3.2}$$

It can be proved that 3.1 $\leq$ 3.2 using combinatorics and probability theory. Figure 3.5 confirms this difference under different k and packet loss rate p. Clearly, the probability is always higher for $n = 2k+1$ than $n = 2k$ given the same k. That means, it is easier for a single node to collect a quorum of messages in a group of $2k+1$ nodes.

3.6 Related Work

To bypass the impossibility of Fischer et al. [31], we can use a randomized algorithm, although the correctness criterion is correspondingly weakened from "must terminate" to "terminates with a probability of 1".

Ben-Or's algorithm [7] is a randomized fault tolerant consensus algorithm for a completely asynchronous system and can withstand a strong adversary. It has a crash fault tolerant variant and a BFT variant. The former can tolerate $f \leq \lfloor \frac{n-1}{2} \rfloor$ crashed processes, which has inspired this work. The BFT version requires $f \leq \lfloor \frac{n-1}{5} \rfloor$, which is less attractive in practice. Bracha's algorithm [9] improves the maximum tolerable faults to $f \leq \lfloor \frac{n-1}{3} \rfloor$ at the cost of using reliable broadcast, which can introduce considerable overhead. Turquois [51] has a novel message validation mechanism to get rid of the reliable broadcast primitive. Meanwhile, it utilizes an efficient message authentication approach and UDP broadcast, making it tailored for wireless embedded systems. The authors did not mention the strong/weak adversary model, but it turns out that Turquois cannot withstand a strong adversary, as discussed in Section 3.4.1. Several works have explicitly addressed the weak adversary model [14, 2], in which the adversary does not know everything about the whole system state. They achieve high efficiency at the cost of having this weakened adversary model. Vavala and Neves also propose a speculative randomized consensus algorithm in a so-called *normal condition* where the adversary model is further relaxed

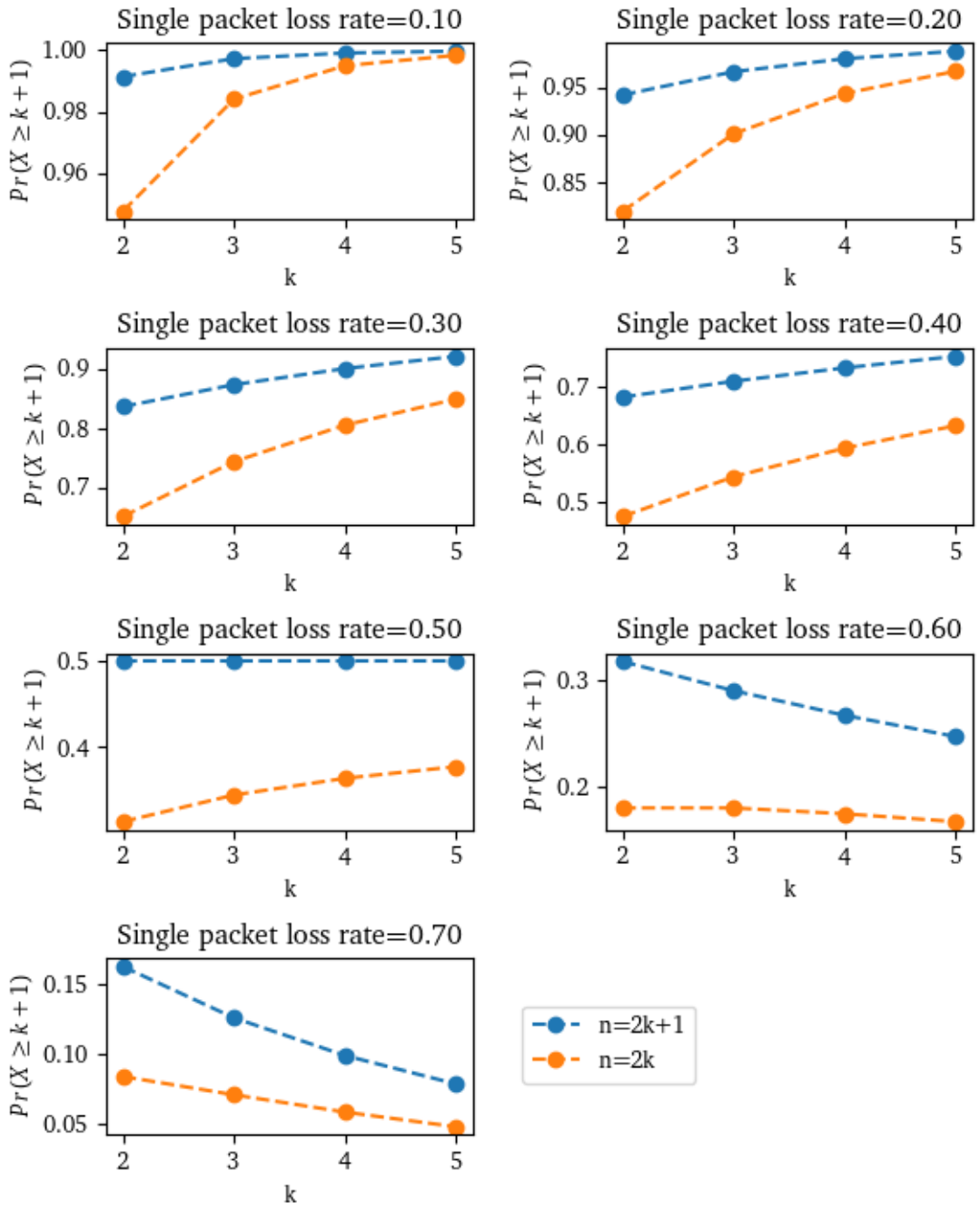


Figure 3.5: The probability that a process successfully receives messages from a quorum with group size $n = 2k$ and $n = 2k + 1$.

compared to the worst case [69]. It is worth mentioning the correctness proof of the crash fault tolerant Ben-Or's algorithm [3]. It gives a good example of how to proof termination under a strong adversary model. But if we take Byzantine faults into account, the proof becomes more complex as we show in this work.

Hybrid fault model [70] is a common approach to increase the maximum tolerable faulty processes and to decrease the complexity of consensus. In this model, a small subset of the system is trusted and cannot be arbitrarily faulty, but can only fail by crashing. One common usage of this subsystem is to prevent double cheating by using one or more monotonic counters for message authentication [43, 71, 37, 6], so that a Byzantine process cannot send contradictory messages to different recipients. But all the above mentioned works are deterministic algorithms and assume partial synchrony. Moreover, they are all designed for state machine replication in data centers. As a result, they tend to be very complex because of some unnecessary features dedicated to their use cases, while they also lack some other features that are needed in wireless embedded systems, such as handling message omission (they rely on TCP to handle it).

Correia et al. [18] discuss the transformation from a crash consensus to Byzantine consensus in hybrid fault model. Although they provide an idea to transform the original Ben-Or's algorithm, above all they require the reliable broadcast primitive. Besides that, the algorithm relies on a failure detector that can eventually find out all Byzantine processes, but the design of this failure detector is unclear. Moreover, the proposed algorithm does not mention a trusted random number generator. We suspect that the Byzantine processes could forever prevent their algorithm from terminating, by manipulating the random value.

3.7 Conclusion

In this chapter, we present TRUSTED BEN-OR, a randomized hybrid fault-tolerant consensus algorithm. It operates in an asynchronous timing model and is resilient against a strong adversary. TRUSTED BEN-OR increases the maximum tolerable Byzantine processes to $\lfloor \frac{n-1}{2} \rfloor$ by utilizing a trusted subsystem in each process. The trusted subsystem encapsulates a monotonic counter authenticator to prevent equivocation, and a trusted coin to generate unbiased random bits. A Byzantine process cannot compromise even its own trusted subsystem. The algorithm is tailored for wireless embedded systems because it does not rely on connection-oriented communication protocols, e. g. TCP, or any complex communication primitives. Neither does it require expensive asymmetric digital signatures. We evaluate TRUSTED BEN-OR on a testbed consisting of 10 Raspberry Pis connected via a wireless ad hoc network. The results show that in most of the cases, the termination latency of TRUSTED BEN-OR is less than that of Turquois – another asynchronous BFT consensus algorithm tolerating only up to $\lfloor \frac{n-1}{3} \rfloor$ Byzantine processes.

4

Deterministic Multi-Value Consensus

The previous chapter presents the randomized consensus TRUSTED BEN-OR that can work perfectly in an asynchronous system. By utilizing a trusted counter, up to $\lfloor \frac{n-1}{2} \rfloor$ Byzantine processes can be tolerated among totally n processes. However, the algorithm can terminate only under certain coincidences, relying on randomness. Although the probability that such coincidences eventually happen is 1, the expected number of rounds until termination increases as the value space expands. That is why TRUSTED BEN-OR is designed only for the binary consensus. To overcome this issue, we also design RATCHETA, a leader-based consensus algorithm that can work with a arbitrarily large value space under a partially synchronous system.

4.1 Multi-Value consensus

Besides the binary consensus, there are many real-life distributed applications that must agree on a value from a big value space rather than $\{0, 1\}$. Imagine the following scenario in a life search and rescue mission: a group (>2) of autonomous life rescue robots are exploring an unknown environment. Every robot has its own sensors and actuators and can make a decision independently. Instead of uncoordinated searching, it would be more efficient if the robots agree on a plan so that each one explores a different region. Apparently, agreeing on a global plan is far beyond a simple yes/no agreement, so the binary consensus does not help in this scenario. We could rely on a centralized coordinator, which however leads to a single point of failure. Any misbehavior of the coordinator would likely break the whole system, thus making it even more counterproductive than the uncoordinated search. For example, if the coordinator crashes or is

unable to communicate, the followers must decide independently, leading to an uncoordinated plan. Even worse, a Byzantine coordinator can make wrong decisions and deliberately mislead the followers. Such Byzantine faults can be caused by software/hardware errors, sensor/actuator malfunctions, malicious attacks, etc. Accordingly, a multi-value Byzantine fault-tolerant (BFT) consensus would be desirable to guarantee that the system can agree on a value from an arbitrary value space — a global search plan in this case.

In this chapter, we propose RATCHETA, a hybrid fault-tolerant consensus algorithm tailored for wireless embedded systems. Similar to TRUSTED BEN-OR, each process is equipped with a trusted subsystem that hosts a message authenticator called BiTrInc to prevent equivocation. BiTrInc can be regarded as an augmented counter authenticator of TRUSTED BEN-OR, and is implemented on top of the Trusted Execution Environment (TEE) of ARM TrustZone [5] as well. It runs in an isolated secure environment that is strongly protected from the remaining system. By strengthening the reliability of each process with BiTrInc, the maximum tolerable faulty processes can be increased to $\lfloor \frac{n-1}{2} \rfloor$ in RATCHETA. Moreover, we address an issue that can be found in several hybrid fault-tolerant consensus algorithms [16, 43, 71, 37], namely the unbounded memory demand and message size. We use a double-counter authentication of BiTrInc to solve this issue.

In summary, the RATCHETA algorithm presented in this chapter possesses the following properties:

- *Optimal fault-resilience*: RATCHETA tolerates $\lfloor \frac{n-1}{2} \rfloor$ Byzantine nodes among n nodes in a partially synchronous system. This is the best bound one can achieve – even if only fail-stop faults are present [22].
- *Bounded memory usage*: it guarantees an upper bound of memory usage and message size in contrast to several related works that tend to require infinite memory.
- *Retransmission-free and tailored for wireless embedded systems*: The algorithm does not assume a low-level communication protocol with packet loss detection and retransmission. Moreover, by using UDP multicast, RATCHETA lowers the communication overhead compared to point-to-point communication.

The third point is also important because in some wireless embedded systems, the reliable message delivery is not preferred or not supported by the low-level protocols. The Basic Transport Protocol (BTP) in the vehicular communication standard [1] is such an example.

We evaluate the algorithm on a testbed consisting of 3-10 Raspberry Pis connected by an ad-hoc wireless network. The results show a promising performance and resilience against Byzantine attacks. Moreover, although not explicitly designed against omission faults, RATCHETA performs well in an unreliable network with certain packet loss rates.

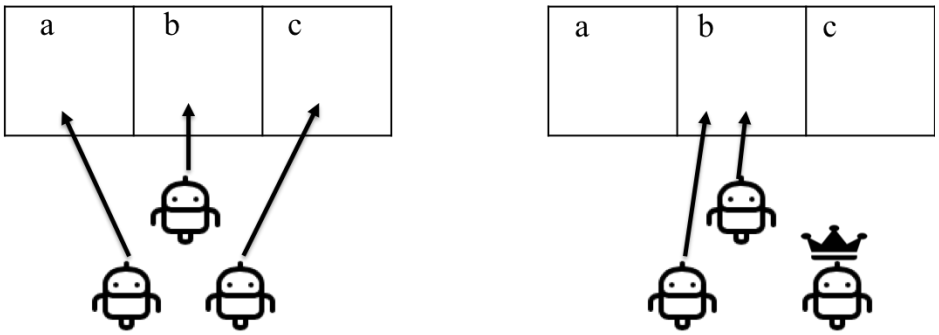


Figure 4.1: Coordination among cooperative robots. The left case is an ideal working plan, where each robot works in a separate region. In a central-coordinator-based solution on the right side, a faulty coordinator could mislead the followers with contradictory plans.

4.2 Use Case Scenario and Preliminaries

Before we explain the details of the consensus algorithm, we firstly show where the algorithm can be applied with a concrete use-case example, and also give a brief introduction about the existing solutions and their limitations.

4.2.1 An Example: Post-Disaster Search and Rescue

We take the same use case as in Section 4.1 to illustrate the underlying problem. A group of automatic robots or UAVs [19] are on a search and rescue mission. Suppose that the whole area to be searched is divided into several regions as shown in Figure 4.1. Ideally each node should search one region that does not overlap with others' (Figure 4.1 left). After a certain time period, each node switches to another region, in case that some faulty nodes do not fulfill their duties. This can be achieved by making all nodes agree on a correct distribution plan, which necessarily requires a consensus algorithm.

As discussed before, relying on a central coordinator to publish the plan is not preferable as it presents a single point of failure. Figure 4.1 (right) shows an extreme case where the faulty coordinator publishes two contradictory plans, letting two followers search the same region. The consensus algorithm should therefore be Byzantine fault-tolerant so that even if a limited number of nodes behave arbitrarily faulty, they cannot prevent the correct ones from agreeing on a valid plan.

We only focus on how to establish an agreement on such a plan. As to the failures during execution, e. g. faulty nodes that do not obey the agreement or maliciously obstruct the others' executions, extra mechanisms are required but they are beyond the scope of this work.

4.2.2 Hybrid Fault Tolerance: Preliminaries and Open Issues

As introduced in Chapter 2.1, every process can be equipped with a trusted subsystem in a hybrid fault model [70] to restrain the ability of a Byzantine adversary. In the previous chapter, we have already seen how TRUSTED BEN-OR has utilized the trusted subsystem to prevent equivocation attacks, meaning Byzantine processes making contradictory statements to different group members. Several leader-based algorithms [16, 71, 37, 6] also adopted this idea and increased the maximum tolerable Byzantine nodes from $\lfloor \frac{n-1}{3} \rfloor$ to $\lfloor \frac{n-1}{2} \rfloor$ in a partially synchronous system.

In order to prevent equivocation, these algorithms rely on "leaving an immutable trace in history" in the trusted subsystem, such as append-only message logs [16] or monotonic counters [43, 71, 37, 6]. Some of them [16, 71, 37] work smoothly in their normal-case operation, where the leader is correct, but during a view-change, i. e. the followers suspect the current leader and want to elect a new one, it becomes problematic. A view-change is not necessarily caused by a faulty leader, but can also caused by message delay in an asynchronous network, so more than one view-changes can take place consecutively before the network becomes synchronous. In each view-change, it is important for every process to provide a proof that it *did not* send anything except for other view-change messages since a certain point in time. Otherwise, a faulty process can arbitrarily conceal its message history to cause disagreement. The aforementioned algorithms provide such a proof by including the whole history of messages since the first view-change in every new message. However, in a partially synchronous system there is no guarantee on how many view-changes will take place until the next stable view can be established. This leads to an unlimited growth of the message history and the message size.

We take the MinBFT protocol as an example and we assume the readers are familiar with its specification described in [71]. Assume three replicas $\{R_0, R_1, R_2\}$ start in a stable view and receive two client requests o_a and o_b, and need to agree on the order of them. According to the protocol, the primary either sends the correct PREPAREs in time, or is suspected by the other replicas and a view-change happens at each follower. Suppose the following situation:

1. During the first period of time ΔT, all messages between R_0 and R_2 are lost (or extremely slow);

2. R_1 is faulty, and it only sends REQ-VIEW-CHANGE messages but nothing else. The REQ-VIEW-CHANGE does not require a counter authentication;

3. At time ΔT, the connection between R_0 and R_2 becomes synchronous, while R_1 crashes immediately.

In this case, R_0 and R_2 can always receive enough REQ-VIEW-CHANGEs with the help of R_1, so they will keep sending VIEW-CHANGEs to move to higher views, but neither is able to establish a new stable view. After ΔT time the system becomes synchronous, and R_0 and R_2 can receive

VIEW-CHANGE messages from each other. The algorithm specification says that every VIEW-CHANGE needs to include "all messages sent by the replica since the latest checkpoint", to make sure that there are no "holes" in the message sequence numbers. As a result, both R_0 and R_2 have to keep all VIEW-CHANGEs they have sent before, and include them in their next VIEW-CHANGEs as a proof of history. Consequently, there must be a long enough period ΔT to exhaust the memory of the replicas. An exponential increase of the view-change timeout can only result in a greater ΔT, but cannot eliminate this issue.

The same problem also exists in other similar protocols, and is only noticed and addressed by a recent work named Hybster [6]. However, Hybster is designed for data centers with TCP networks that support a reliable message delivery. If such support is unavailable, the reliable messaging should be explicitly tackled by the consensus algorithm, which will increase the computation and network overhead.

In this work, we solve the problem of unbounded memory demand by using two counters in parallel instead of one. Furthermore, we do not assume any reliable messaging mechanisms for packet loss detection and retransmission.

Remark 4 *This unbounded memory issue in the aforementioned algorithms [16, 71, 37] actually originates from their timing assumptions in the partially synchronous system. The view-changes are triggered by timeouts. Therefore, if a process is partitioned from the other participants for a long time, it may possibly jump over several views without receiving any messages. When the system partition is resolved and this process can connect to others, it has to know what has happened during those views it has missed, so other processes must keep their histories to convince it to safely establish a stable view.*

A fully asynchronous algorithm such as TRUSTED BEN-OR*, on the contrary, does not have this issue, because there is no timing assumption in the algorithm design. Each process proceeds to the next round only if it has received messages from a quorum, so it will not overstep any round. Before entering a round, the process has already known enough information about the previous rounds.*

4.3 BiTrInc: the Trusted Counter Authentication

Now we introduce the trusted message authenticator BiTrInc. It is similar to the authenticator of TRUSTED BEN-OR (Section 3.2.2). BiTrInc also runs in a trusted subsystem. Unlike TRUSTED BEN-OR, it uses a pair of counters to prevent equivocation, instead of one counter.

To understand why two counters are better than one in a partially synchronous system, we briefly explain the cause of equivocation attack in these BFT consensus algorithms. Most of these algorithms can be abstracted as a series of voting rounds. Within one round, the processes vote to decide on a common value. Between two consecutive rounds, the processes have to exchange information to find out if any correct process has already decided on a value previously. If so, it is

not safe to vote for a different value in the next round. From this abstraction we can observe two possibilities of equivocation:

- Within one voting round, a Byzantine process sends different messages to different recipients.
- In an earlier voting round, a Byzantine process has sent some message(s), leading a correct process to deciding on a value. In a later round, the same Byzantine process conceals this fact.

Both cases can result in contradictory decisions among the correct processes. We denote the first case as the *in-round equivocation*, and the second as the *cross-round equivocation*. The in-round equivocation can be described as "I say x to A and say y to B", while the cross-round equivocation is "I said x to A and then I say 'I did not say anything' to B".

In order to prevent the two types of equivocation, we have devised BiTrInc, a counter-based message authentication module inspired by TrInc [43]. It can create multiple *authenticators*, and each of them contains a pair of monotonic counters. Although TrInc can also use multiple counters, it claims "anything done with multiple counters can be done with a single counter".[1] However, we have seen the drawback in 4.2.2 if we are too greedy and use one single counter to solve two problems simultaneously. Especially the cross-round equivocation handling might cause an infinitive growth of memory usage.

BiTrInc can solve the unbounded history issue and simplify the design of the consensus algorithm, by using the two counters to individually handle the two types of equivocation. One *non-decreasing* counter is used to record the latest voting round to prevent the cross-round equivocation. The other *strictly increasing* counter gives a process only one chance to vote in every round, so the in-round equivocation becomes impossible.

The APIs of BiTrInc are listed below, and the notations are explained in Table 4.1. The counter u_1 is non-decreasing, while u_2 is strictly increasing. Other details such as remote attestation as well as key distribution and storage remain the same as for TrInc and are omitted.

- `CreateCounter()`: increases cid^{max} by one, and creates a new counter authenticator with $cid = cid^{max}$; initializes its counter pair (u_1, u_2) to $(0, 0)$; returns cid to the caller.
- `authenticate(`$m, cid, (u_1', u_2')$`)`: requires $u_1' \geq cid.u_1 \wedge u_2' > cid.u_2$; sets $cid.u_1$ to u_1' and $cid.u_2$ to u_2'; returns the authentication code $sid||cid||u_1'||u_2'||HMAC(K_{cid}, m||sid||cid||u_1'||u_2')$.
- `verify(`$m, sid||cid||u_1||u_2||h$`)`: verifies if h is generated by $HMAC(K_{cid}, m||sid||cid||u_1||u_2)$; return *true* or *false*.
- `DeleteCounter(`cid`)`: deletes counter authenticator cid if it has been created and not yet deleted.

Table 4.1: Notation used to explain BiTrInc

Notation	Meaning
sid	the unique identifier of the BiTrInc
cid^{max}	the highest counter authenticator ID assigned so far
cid	the identifier of the counter authenticator
K_{cid}	the secret key of counter authenticator cid
(u_1, u_2)	the tuple of values of the counter authenticator
m	the original message to be authenticated
$\|\|$	the concatenation operator

Different from TrInc, BiTrInc exposes the sid, cid and both counter values to the verifiers. This modification decouples the authentication from the verification, and does not require a record for the most recent authentications as in TrInc.

For the same reason as in TRUSTED BEN-OR (see Chapter 3.2.2). The authenticator can use symmetric encryption algorithms such as HMAC instead of asymmetric digital signature for message authentication. It is much more efficient, while still guaranteeing the same non-repudiation feature, because of the trusted subsystem.

To avoid a faulty node presenting different counter authenticators to different processes in one agreement instance, BiTrInc also uses a monotonically increasing cid to identify each authenticator, which is a one-to-one correspondence to an agreement instance. Compared to only one agreement instance [22, 52, 51], if multiple instances are considered [17], a unique instance ID is necessary to distinguish them. The application has to define the mapping rule from instance ID to cid, e.g. the easiest way is to directly use cid as the instance ID. It is also the application's duty to decide if and when it should start an agreement instance, and with which ID. In SMR, for example, the instance ID is the sequence number of the ordered operation. In time-related applications, such as the life-rescue group from Section 4.2, it is more complex. Suppose that every node periodically starts an agreement instance at the pre-configured time $t_1, t_2, \ldots$. If the correct nodes have synchronized their clocks e. g. via GPS, which is commonly available on the platforms of these application domains [19], they can then infer the correct instance ID from their clocks, and a correct node can ignore an outdated or advanced ID. Since this issue does not belong to the consensus algorithm and is application-dependent, we omit the detailed discussion here.

4.4 RATCHETA Algorithm Design

RATCHETA is a leader-based algorithm. The algorithm specification is presented in Algorithm 4.1. Now we give a brief explanation to the algorithm.

[1]Though they might use multiple counters for totally different purposes, for example in the TrInc based A2M [43].

The algorithm execution contains multiple *rounds* with successive round numbers. The change of rounds is driven by timeouts. In each round there is a unique *coordinator* that can be identified by every process. This leader- and round-based idea is common in fault-tolerant consensus, e. g. Paxos [39], PBFT [15] and MinBFT [71]. The role of the coordinator is rotated in a round-robin manner from round to round, so that eventually a correct process can become the coordinator. One example to achieve this is to use the round number modulo the group size (n) to calculate the ID of the coordinator, as in PBFT [15].

In RATCHETA, each round consists of 3 *phases*. In phase 1, every process broadcasts a *round-change* message about its current status, which is collected by the designated coordinator.

If the coordinator has collected enough round-change messages, it selects a value satisfying the `can_prove` function and *proposes* the value in phase 2. The proposal should either be based on a previous proposal from a round r^{base}, or be chosen from a set of initial values if no previous proposals exist. In the latter case, there can be different strategies to choose a value from the initial values — depending on the validity requirement. This leads to different implementations of the `can_prove` function. For example, Algorithm 4.2 requires that the value is proposed by at least one process, namely the weak validity. Algorithm 4.3 requires that the value is the median of all the initial values. As we will prove later, this guarantees the median validity if $n > 3f$. More details about the `can_prove` function will be explained in Section 4.4.2.

In phase 3, if a follower receives the proposal from the coordinator, it sends a *confirmation* after checking the validity of the proposal using the same `can_prove` function. Once a process learns that a *quorum* has confirmed the proposal, it can decide on that value. A quorum Q is defined as a subset of all the processes Π containing at least $\lceil \frac{n+1}{2} \rceil$ processes. This guarantees that any two quorums intersect with at least one process. All quorums constitute the family of sets $\mathcal{Q}$, which is referred to in the algorithm. For convenience, we sometimes say "a quorum of messages", which means that a set of messages sent by a quorum of processes.

Each process also maintains a set $lACK$ denoting the late-acknowledged rounds. This is used to ensure liveness and will be discussed in Section 4.4.3.

Despite the existence of Byzantine processes, the use of BiTrInc can preclude all kinds of equivocation, as we will show later. Besides, even if a process has decided, it will continue to execute the algorithm (but will not decide again), in case there are others who need its help to decide.

A message in RATCHETA authenticated by BiTrInc is denoted as $\langle message \rangle_{\tau(p,u_1^\ell,u_2^h)}$, where $\tau(p, u_1^\ell, u_2^h)$ is the authentication code created by the authenticator of process p with the counter values (u_1^ℓ, u_2^h). The two counters are named ℓ and h and their values are u_1 and u_2, respectively. Counter ℓ is non-decreasing, while h is strictly increasing. For simplicity, we omit the cid as it remains the same for each individual correct process during the algorithm execution.

Figure 4.2 depicts the fault-free case with three processes, where all processes are correct and all messages are delivered in time (the use of the trusted counter will be discussed later). The running of the algorithm can be divided into three steps:

1. The quorum size should be 2, so the coordinator can propose after receiving any two initial values, in this case the values of p_1 and p_2.

2. When p_2 (or p_3) receives the proposal, it feels satisfied with it. p_2 together with the coordinator can already form a quorum, so p_2 immediately decide. Meanwhile, it will send a confirm message.

3. Once the coordinator has collected enough confirmation, it can also decide. Here with only three processes, a coordinator only needs to wait for another one confirm message, e. g. from p_2.

It is worth noting that $n = 3$ is just a special case, where a follower can immediately decide after receiving a valid proposal, because only in this case, the quorum size is 2. If $n = 4$ for example, as shown in Figure 4.3, each follower has to wait for one more confirmation from another follower to decide.

The following sections will explain the design of RATCHETA in details.

4.4.1 Unique Messages per Round

The strictly increasing counter h is used to prevent the in-round equivocation, namely the faulty processes sending contradictory messages in the same round.

It is to be observed that a correct process will create at most two messages in each round. One is in the phase 1 to report its status. The other is sent either in the phase 2 or 3, corresponding to either the proposal of the coordinator or the confirmation of the follower, respectively. Accordingly, we categorize the messages into two classes:

Definition 10 *A message sent in phase 1 of every round is called a **round-change message**. A message sent in phase 2 or 3 is called a **vote message**. We say a process **votes for** v if it is the coordinator and sends the proposal (line 16), or it is the follower and sends the confirmation (line 34) of value v in round r.*

We use the counter h to bind one message to a specific round r and a message type. More specifically, a message m is bound to a counter value $u^h = [r|b]$ where the most significant bits represent the round number r, while the last single bit indicates the message type: $b = 0$ for round-change and $b = 1$ for vote message.

Because the counter h is strictly increasing, it ensures that one process can only authenticate a unique round-change and a unique vote message per round. Formally:

Algorithm 4.1: RATCHETA algorithm

```
1  Initialization:
2      CreateCounter()
3      vote ← initial value of p
4      r ← 0
5      r^last ← 0
6      PROPOSAL^last ← ⊥
7      CERT^last ← ∅
8      lACK ← ∅                                          / late-acknowledged rounds /

10 Phase 1:                                              / round-change /
11     r ← r + 1
12     broadcast(⟨vote, r^last, lACK⟩_τ(p,⌈r^last⌉^ℓ,⌈r|0⌉^h))
13     if p is coordinator of round r do
14         wait until p has collected a set 𝒱 of messages s.t.
               ∃v', r^base : can_prove(v', r^base, r, 𝒱)        /* ref. 4.4.4 */
15         vote ← v'
16         PROPOSAL^last ← ⟨vote, r^base⟩_τ(p,⌈r⌉^ℓ,⌈r|1⌉^h)
17         CERT^last ← 𝒱
18         r^last ← r

20 Phase 2:                                              / propose /
21     if p is coordinator of round r
22         if r^last = r do
23             broadcast(PROPOSAL^last || CERT^last)
24     else do
25         wait until (proposal || 𝒱) received from coordinator q or timeout   /* ref. 4.4.4 */
26         if proposal = ⟨vote', r^base⟩_τ(q,⌈r⌉^ℓ,⌈r|1⌉^h) and can_prove(vote', r^base, r, 𝒱) do
27             vote ← vote'
28             PROPOSAL^last ← proposal
29             CERT^last ← 𝒱
30             r^last ← r

32 Phase 3:                                              / confirm or round-forward /
33     if p is not coordinator of round r and r^last = r do          / confirm proposal /
34         broadcast(⟨vote, r⟩_τ(p,⌈r⌉^ℓ,⌈r|1⌉^h) || PROPOSAL^last || CERT^last)
35     else if r^last > 0 do                             / forward last seen proposal /
36         broadcast(NULL || PROPOSAL^last || CERT^last)
37     lACK ← ∅
38     foreach received message (m' || proposal || 𝒱) do          /* ref. 4.4.4 */
39         if not decided and has collected ≥ (⌈(n+1)/2⌉) different m' or proposal signed with
               τ(∗, ⌈r⌉^ℓ, ⌈r|1⌉^h) voting for vote do
40             decide vote
41         if proposal = ⟨vote', r^base⟩_τ(∗) is from round r' > r^last and   / late-acknowledge /
               can_prove(vote', r^base, r', 𝒱) do
42             lACK ← lACK ∪ {(vote', r')}
```

Algorithm 4.2: The function to verify a proposal certificate satisfies weak validity.

```
1  Function can_prove(vote, r^base, r^cur, 𝒱)
2      if 𝒱 contains ⌈(n+1)/2⌉ different round-change messages of round r^cur and
3        ∀m' = ⟨*, r', *⟩_τ(*) ∈ 𝒱 is authenticated with counter pair u_1^ℓ = [r'] ∧ u_2^h = [r^cur|0]
               and
4        Case A: r^base = 0
5              if ∀m' ∈ 𝒱: m' = ⟨*, 0, *⟩_τ(*) and vote is chosen from 𝒱
6                  return true
7        Case B: r^base > 0
8              if ∀m' = ⟨*, r', *⟩_τ(*) ∈ 𝒱 : r' ≤ r^base and
9                  there are at least f + 1 such messages m' ∈ 𝒱 that
                       either m' = ⟨vote, r^base, *⟩_τ(*)
                       or (vote, r^base) ∈ m'.lACK
10                 return true
11     return false
```

Algorithm 4.3: The function to verify a proposal certificate satisfies median validity (requires $n > 3f$).

```
1  Function pick_median(𝒱)
2      A ← the values carried by the round-change messages 𝒱, duplication is allowed
3      sort A
4      c ← ⌊(A.length − 1)/2⌋
5      return A[c]

7  Function can_prove(vote, r^base, r^cur, 𝒱)
8      if ∀m' = ⟨*, r', *⟩_τ(*) ∈ 𝒱 is authenticated with counter pair u_1^ℓ = [r'] ∧ u_2^h = [r^cur|0]
           and
9        Case A: r^base = 0
10             if 𝒱 contains (n − f) different round-change messages and
                   ∀m' ∈ 𝒱: m' = ⟨*, 0, *⟩_τ(*) and vote = pick_median(𝒱)
11                 return true
12       Case B: r^base > 0
13             if 𝒱 contains ⌈(n+1)/2⌉ different round-change messages and
                   ∀m' = ⟨*, r', *⟩_τ(*) ∈ 𝒱 : r' ≤ r^base and
14                 there are at least f + 1 such messages m' ∈ 𝒱 that
                       either m' = ⟨vote, r^base, *⟩_τ(*)
                       or (vote, r^base) ∈ m'.lACK
15                 return true
16     return false
```

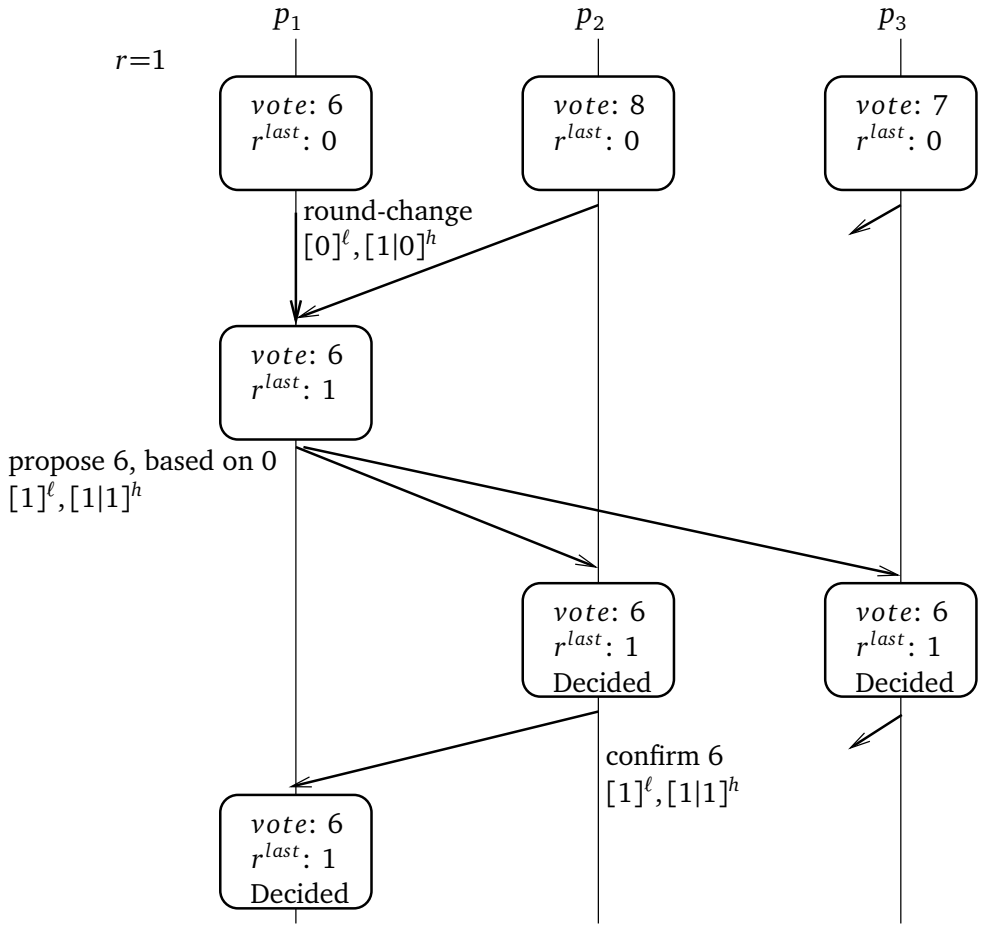


Figure 4.2: Fault-free case with three processes. Not all messages are listed here. Short arrows mean that they arrive too late thus have no effect on the recipients.

Lemma 17 *In each round r, the process p can only create at most one round-change message $\langle vote, r^{last}, lACK \rangle_{\tau(p,[r^{last}]^{\ell},[r|0]^{h})}$. Depending on p is the coordinator in this round or not, it can only create at most one vote message $\langle vote, r^{base} \rangle_{\tau(p,[r]^{\ell},[r|1]^{h})}$ or $(\langle vote, r \rangle_{\tau(p,[r]^{\ell},[r|1]^{h})}$.*

Furthermore, because every round has a unique coordinator that is known to every process, it can be inferred that in each round there is at most one valid proposal:

Corollary 18 *In each round, there is at most one proposal with the correct authentication.*

Figure 4.4 shows how the counter h can prevent a faulty coordinator from cheating. The coordinator p_1 has firstly proposed 6 to p_3, letting p_3 to decide 6. By doing this, p_1 has to set the value of h to $[1|1]$. If it later wants to propose a different value to p_2, it cannot bypass BiTrInc to authenticate the new proposal. Thus, a different proposal should be detected and discarded by p_2.

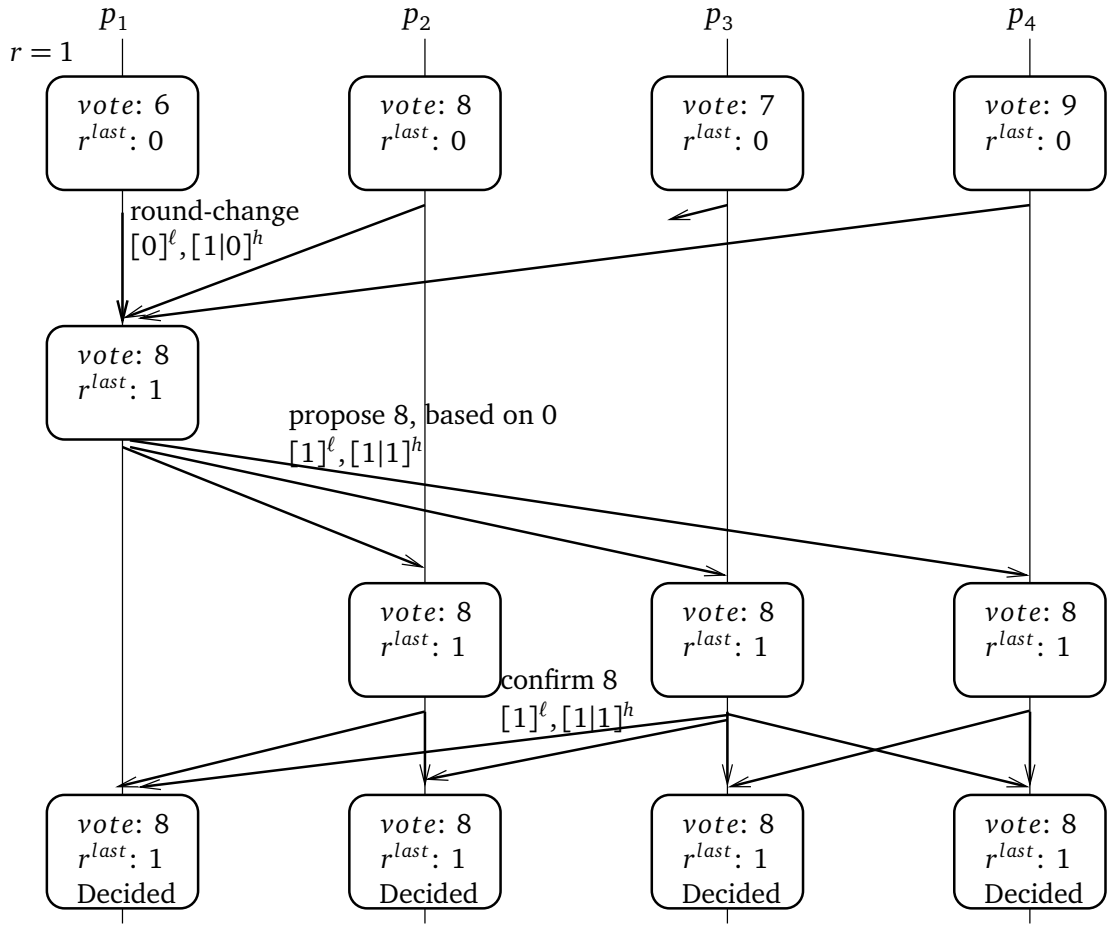


Figure 4.3: Fault-free case with four processes. The `can_prove` function is implemented as in Algorithm 4.3 for median validity.

4.4.2 Proposal Certificate

Because of potential faulty coordinator and communication failures, the protocol has to continuously change to new rounds. However, it could happen that some correct processes have already decided value v_1, but the malicious processes want to hide this information and support another proposal v_2. We use another non-decreasing counter ℓ to prevent this kind of cross-round equivocation. We say a round r is *active* to process p, if p has voted in round r. Accordingly, p is *from* round r, if r is its latest active round. To ensure that p cannot conceal its active round r, the value u^{ℓ} should be set to $[r]$ if it has voted in round r. The counter ℓ is not strictly increasing because a process is not necessarily active in every round. To understand why this can happen, recall that round-change is driven by timeouts. If the system is still in the asynchronous period, and a process p did not receive any message in a considerably long time window, it could experience several rounds in which it cannot vote, but can only send round-change messages. As a result, u^{ℓ} would remain unchanged for several consecutive rounds.

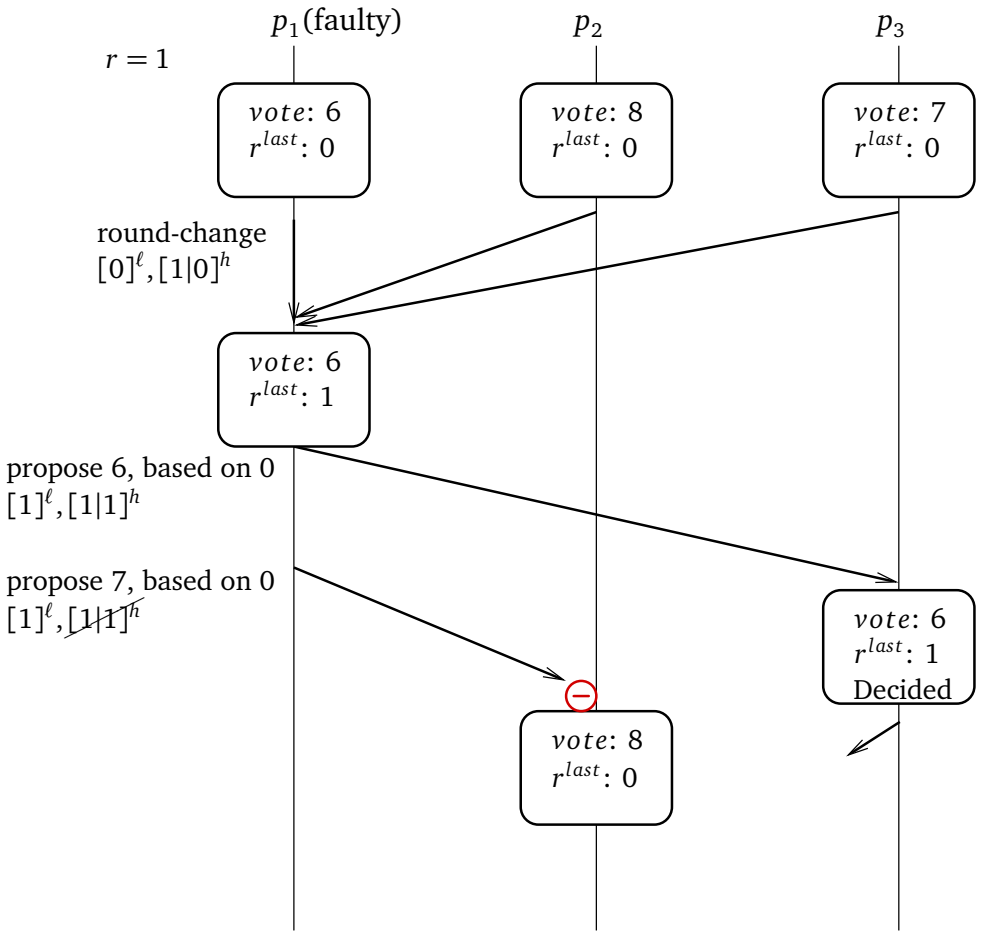


Figure 4.4: Use counter h to prevent a faulty coordinator from in-round equivocation.

In summary, the messages sent in round r by process p have the following authentications:

- Round-change message (line 12) with $\tau(p, [r^{last}]^\ell, [r|0]^h)$;
- Vote message (line 16 or 34) with $\tau(p, [r]^\ell, [r|1]^h)$.

Because both counters are monotonic and trustworthy, we have the following invariant to make sure that faulty processes cannot conceal their history:

Lemma 19 *If a process p votes in round r, then for any later round $r^+ > r$, there exists no valid round-change message $\langle m \rangle_{\tau(p,[r^-]^\ell,[r^+|0]^h)}$ letting it to claim that it is from a previous round $r^- < r$.*

Proof. If p has not used BiTrInc to sign the round-change message $\langle m \rangle_{\tau(p,[r^-]^\ell,[r^+|0]^h)}$ for any $r^+ > r$ and $r^- < r$ before, after it votes in round r, it must set its counter pair to $([r]^\ell, [r|1]^h)$. Because u^ℓ is non-decreasing, it can never set it back to $[r^-]^\ell$ for any $r^- < r$. If p has already signed a round-change message $\langle m \rangle_{\tau(p,[r^-]^\ell,[r^+|0]^h)}$ for some $r^+ > r$ and $r^- < r$ (only a faulty process will do this before it enters round r), then it cannot vote in round r because counter u^h is monotonically increasing. ∎

To generate a valid proposal in round r^+, the coordinator has to collect a set of round-change messages from a quorum, called *proposal certificate based on* r, which can fulfill the following two properties:

Property 1 *Every process in the quorum is from some round no later than the base round r (line 8 of Algorithm 4.2 or line 13 of Algorithm 4.3).*

Property 2 *At least $f+1$ processes accept the correctness of the base round r: they are either directly from r, or can late-acknowledge r (explained in the next subsection, line 9 of Algorithm 4.2 or line 14 of Algorithm 4.3).*

The first requirement ensures that a majority has not voted for anything between r and r^+. The second one implies the correctness of r because at least one correct process accepts it. For any r^+ based on r, the coordinator of r^+ can only propose the same value voted for in r, which should be unique because there can only be one proposal in each round. As for the corner case $r = 0$, meaning the majority has never voted before, the coordinator picks a value satisfying the `can_prove` function (Case A of Algorithm 4.2 or 4.3). The coordinator announces this r in its proposal, and appends the proposal certificate as well, so that the followers can invoke the `can_prove` function to check the aforementioned two requirements.

Figure 4.5 gives an example how a malicious process can conceal its history to cause equivocation, and how this can be prevented by the non-decreasing counter ℓ. In round 1, p_2 has voted for $v = 6$, letting p_1 to decide, while p_3 did not see the messages from the others because of the network asynchrony. In round 2, p_2 will "roll back" its state, as if it has never voted before. Then together with $p3$, it can compose a valid proposal certificate based on round 0, letting p_3 to decide a different value. However, the counter ℓ of BiTrInc of p_2 is already set to $[1]$, and cannot be reset to $[0]$. As a result, p_2 can never collect a quorum of round-change messages from round 0.

4.4.3 Active Round-Forwarding and Late-Acknowledgment

A process might miss some valid proposals due to connection failures, or because a faulty coordinator sends its proposal to only a subset of the group. As a result, the correct processes might be from different rounds, and no $f+1$ ones can confirm the correctness of any round together. An example is shown in Figure 4.6. Here, p_1 has proposed in round 1, and the proposal is delivered to neither p_2 nor p_3. In round 2, p_2 can still collect a valid proposal certificate based on round 0, so it will propose a value. Unfortunately, this proposal is lost again. Now in round 3, the three processes are from three different rounds. As a result, the coordinator cannot collect a valid proposal certificate.

To solve this issue and ensure the liveness, each process has to let the others know about the rounds they possibly missed. It stores the proposal and the corresponding certificate of its last

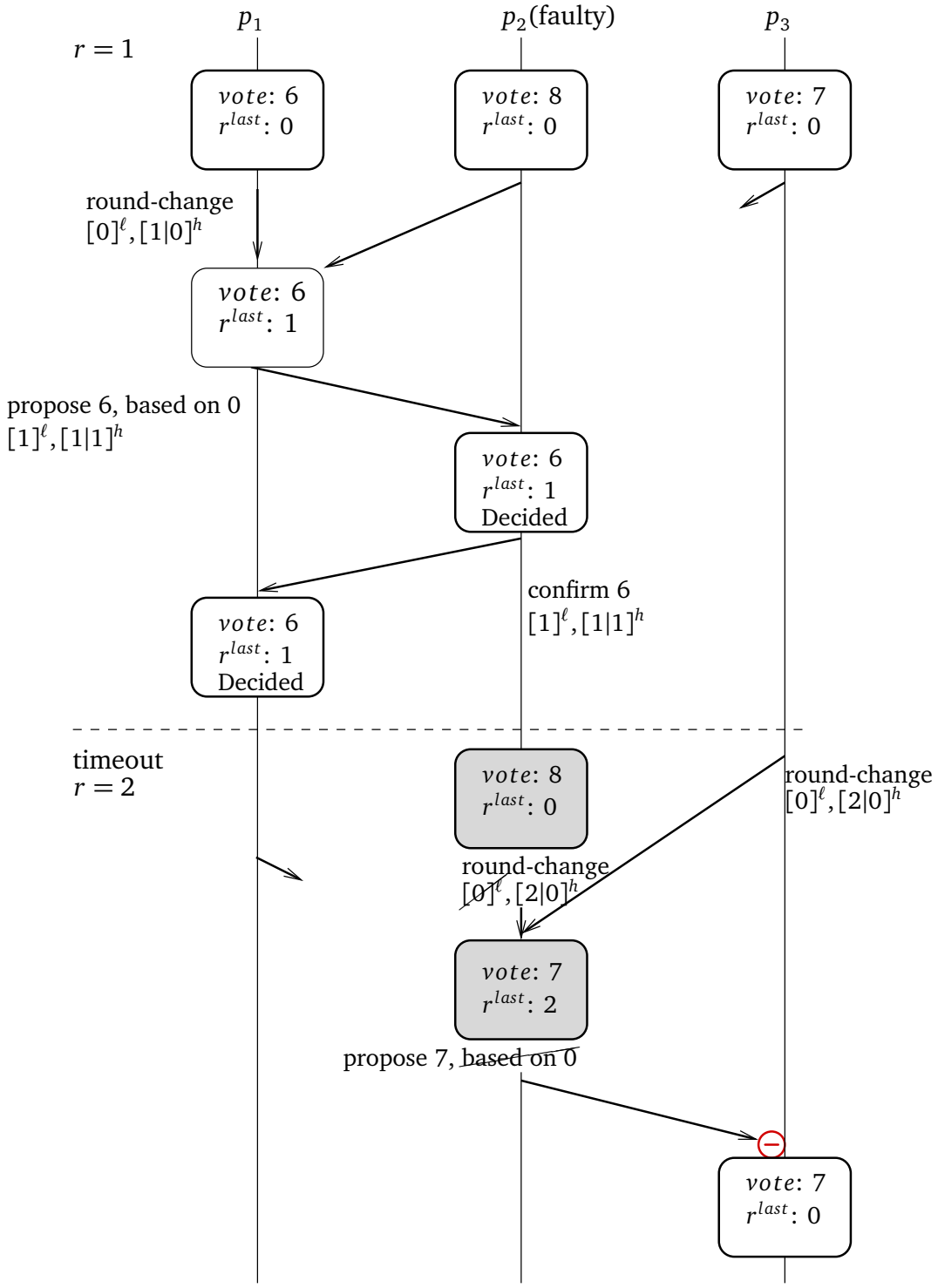


Figure 4.5: Use counter ℓ to prevent a faulty process from concealing its history.

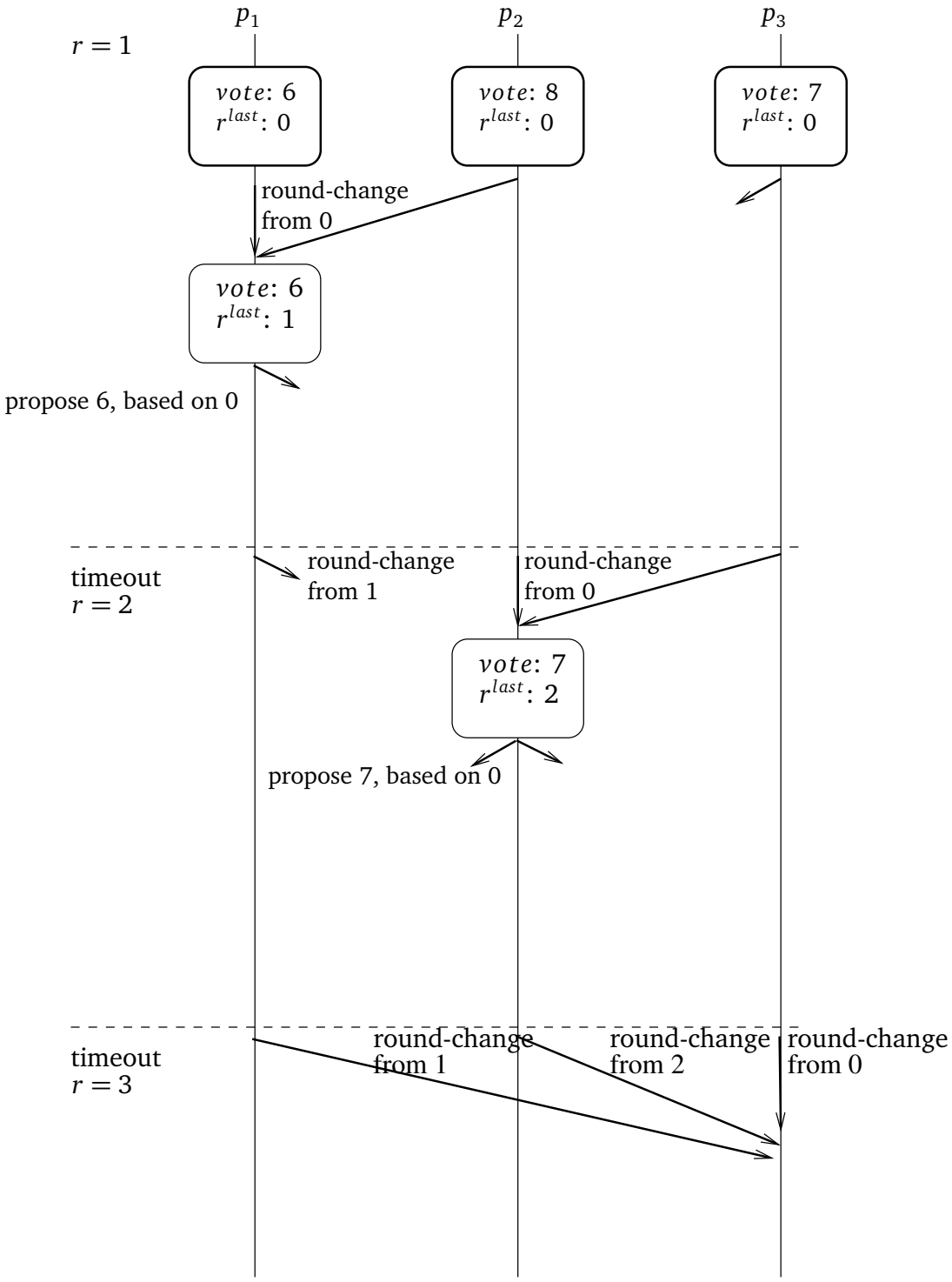


Figure 4.6: A system ends up with all processes from different rounds.

active round, and forwards them to others in phase 3 of each round. If a process receives a valid proposal of round r' greater than its own last active round, it will *late-acknowledge* this proposal. It neither sends a vote message, nor increases its counter. Instead, it only puts a tuple $(vote', r')$ into the set $lACK$, meaning that it accepts the correctness of r'. The set $lACK$ will be included in the next round-change message.

This late-acknowledgment mechanism is shown in Figure 4.7. We consider the same system that gets stuck in Figure 4.6. In round 3, the coordinator cannot make any proposal. However, by the end of this round, p_1 and p_2 will forward their last active rounds to others. As a result, p_1 is informed about the proposal of round 2, in which it did not take part. It will verify the proposal and acknowledge it. Similarly, p_3 will acknowledge both proposals of round 1 and 2. Then in round 4, the coordinator can successfully make a proposal based on round 2.

4.4.4 Implementation of Rounds

As mentioned before, round changes are driven by timeouts. We use two timeouts to implement the round model, so that the algorithm will not be blocked at line 14, 25 and 38. The first timeout terminates phase 1 and 2. This timeout is necessary because it leaves a margin for the processes to forward their last active round. Otherwise, a faulty coordinator of round r can deliberately send a proposal only to f correct processes at a very late time, letting them actively vote in the current round, while the others are left behind in an older round. This prevents the processes from assembling a valid proposal certificate in the next round, because no $f + 1$ processes can confirm the correctness of r together. When the first timeout expires, a process should stop voting in this round, and start to late-acknowledge its most recent received proposal. The second timeout will trigger a process to enter a new round. The communication pattern is illustrated in Figure 4.8.

The two timeout intervals can be initialized with empirical values, and gradually increase round by round. This ensures that even if the clocks of different processes have limited drifts, eventually all processes can stay in the same round.

4.5 Correctness Proof

In this section we will give a proof of correctness with respect to the four properties:

- Agreement;
- Termination;
- Validity;
- Limited memory usage and message size.

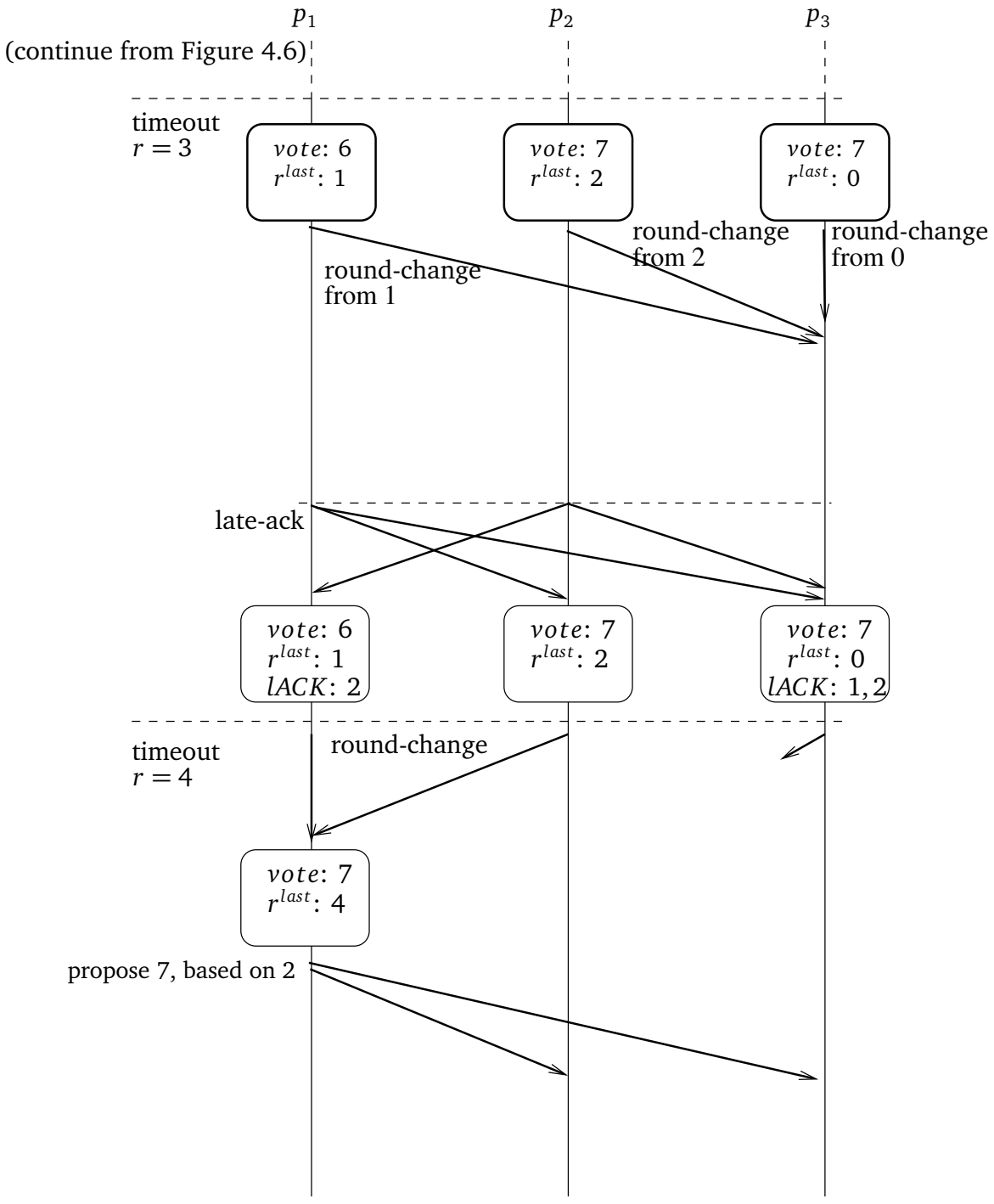


Figure 4.7: Late-acknowledgment to ensure the liveness.

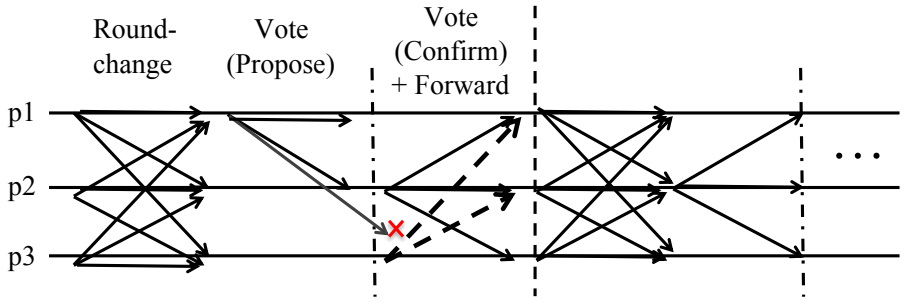


Figure 4.8: Communication pattern of RATCHETA. Note that $p3$ does not accept the proposal of $p1$ because it is determined to stay inactive in this round. It only forwards the information about its last active round as drawn in dashed lines.

4.5.1 Agreement

Similar to TRUSTED BEN-OR, we will prove that if any correct process decides a value, no other correct processes can decide a different value. We do this step by step with several lemmas.

Lemma 20 *If a correct process p votes for v in some round, then v must be proposed in a correctly certified proposal.*

Proof. Correct processes always follow the algorithm specification. If p is the coordinator of the round, its vote for v indicates that it has collected a certificate and proposes v. If p is not the coordinator, it only votes for v if it has successfully verified the proposal certificate. ∎

According to Corollary 18:

Corollary 21 *If two correct processes vote in the same round, they must vote for the same value.*

Lemma 22 *If a correct process decides v in round r, then v must be proposed in a correctly certified proposal.*

Proof. A correct process decides v indicates that a quorum of processes vote for v. Among the quorum, there is at least one correct process, so the proposal with v must be correctly certified according to Lemma 20. ∎

Therefore we can conclude:

Lemma 23 *If two correct processes decides in the same round, they decide the same value.*

Proof. This can be directly proved by the previous lemma and Corollary 18: a correct process decides v only if v is proposed in that round, and there is at most one valid proposal in each round. ∎

Lemma 23 ensures that within the same round, correct processes will not decide differently. Now we consider the situation where they decide in different rounds. The idea is to prove that once the first correct process decides, all proposals in future rounds must be with the decided value. Firstly we have the following lemma:

Lemma 24 *If a correct process decides in round r, there exists no valid proposal of round $r^+ > r$, which is based on a previous round $r^- < r$.*

Proof. If a correct process decides in round r, a quorum of processes must have voted in round r. We prove by contradiction. Assume in a later round, there is a valid proposal based on round r^- This means that a quorum of processes are all from rounds no later than r^-, according to Property 1 of the proposal certificate. Any two quorum must intersect with one process, so this process has voted in round r, and created a round-change message claiming it is from round $r' \leq r^- < r$. This causes a contradiction as it violates Lemma 19. ∎

This lemma explains why we need a dedicated non-decreasing counter to record the last voting round of each process. In this way, once a correct process decides in round r, all following correctly certified proposals can only be based on r or later rounds. Then there is the following lemma:

Lemma 25 *If a correct process decides v in round r, and if there is a correctly certified proposal in a following round $r^+ > r$, its proposed value must be v.*

Proof. We prove by induction.

Base case: if in round $r^+ = r + 1$ there is a correctly certified proposal, it must be based on some round $r' < r^+$. Because of Lemma 24, $r' \geq r$ must hold, so $r' = r$.

Induction assumption: assume all correctly certified proposals from round r to an arbitrary round $r^+ > r$ have proposed v. Note that not every round has to have a correct proposal.

Induction step: denote r^{++} the smallest round which is after r^+ and has a correctly certified proposal, then this proposal must be based on some round r' where $r \leq r' < r^{++}$. The Property 2 of the proposal certificate requires that at least $f + 1$ have voted in r', among which there is at least one correct process. Recall Lemma 20 that a correct process only votes if the proposal in that round is correctly certified. Because there is no valid proposal between r^+ and r^{++}, so the base round r' must satisfy $r \leq r' \leq r^+$, and the proposal in r' must be correctly certified. According to the induction assumption, the proposed value in r' is v, so in r^{++} the same value v must be proposed. ∎

Now we can conclude the proof of safety:

Theorem 26 *No two correct processes decide different values.*

Proof. Denote r the smallest round number in which a correct process decides, and denote v the decision. No correct processes decide before r. If there is another process also decides in r, it must also decide v because of Lemma 23. If another process decides v' in a later round $r^+ > r$, then v' is in a correctly certified proposal of round r^+ (Lemma 22). Finally from Lemma 25, we know that any correctly certified proposal can only have value v, which is the same as round r, so $v = v'$. ∎

4.5.2 Termination

The correctness regarding termination relies on further assumptions of the timing model. Otherwise, in a fully asynchronous system, this cannot be guaranteed [31]. We adopt the same basic round model and partial asynchrony condition of Dwork et al. [22]. A basic round is synchronized among all processes. Each round consists of a send sub-round, a receive sub-round and a computation sub-round, so it corresponds to a phase in RATCHETA protocol. To avoid confusion with our own definition of round, we denote such a basic round as a phase in the following proof. A phase is closed, meaning that a process will not deliver or process a message from any previous phase.

The partial synchrony assumes there is a *Global Stabilization Time (GST)* with respect to phases. Before the GST, there is no guarantee that a message will be delivered in the same phase, even between correct processes. From the GST on, all messages sent by correct processes will be delivered and processed by all correct recipients. Nevertheless, the GST is not known by the processes in advance or during the execution.

With this notation, we can prove the termination of RATCHETA. Firstly, there is the following lemma:

Lemma 27 *Eventually, there is a round r subject to:*

- *In round r, every message from correct senders are delivered and processed by any correct recipient.*
- *In round $r+1$, every message from correct senders are delivered and processed by any correct recipient.*
- *The coordinator of $r+1$ is correct.*

Proof. According to the partially synchronous model, after GST, all messages can be delivered and processed in time. Because of the coordinator rotation mechanism, eventually such a round will occur, which is after GST, and the coordinator of its consecutive round is correct. ∎

Theorem 28 *Every correct process eventually decides.*

Proof. We show that if a round r described in Lemma 27 occurs, all processes can decide no late than round $r+1$. There are two cases to be considered:

Case 1: there is no correct process ever voted before round $r+1$. Because all the correct round-change messages can be delivered by the correct coordinator, these $(n-f)$ messages can constitute a proposal certificate based on round 0 to bypass either version of `can_prove` function (Algorithm 4.2 and 4.3). The coordinator can then send this proposal. The other correct processors will confirm it accordingly. All these vote messages can be delivered in time, thus all correct processors can decide by the end of round $r+1$.

Case 2: one or more correct processes have voted between round 1 and r (inclusive). All correct processes will not change their last active round after they have started phase 3 of round r and before they enter round $r+1$. Among all correct processes , assume p has the highest last active round number $\hat{r}$ during phase 3 of round r. For any other correct process, either it has the same last active round $\hat{r}$, or it can receive the message from p, and late-acknowledge $\hat{r}$. In round $r+1$, the coordinator can at least collect a proposal certificate based on $\hat{r}$ and propose it. Similarly, all vote messages can be delivered in time and all correct processors can decide by the end of round $r+1$. ∎

4.5.3 Validity

Validity is the relation between the decided value and the initial values of processes, but the existence of Byzantine faulty processes makes the issue complicated, because a Byzantine process can cheat about its initial value. To simplify the discussion, we define the initial value as follows:

Definition 11 *The initial value of a process p, no matter faulty or not, is the $vote$ appeared in a round change message based on round 0: $\langle vote, 0, lACK\rangle_{\tau(p,[0]^{\ell},[r|0]^{h})}$.*

Apparently, this definition is trivial to correct processes, because they must send their initial values in round 1. As to the faulty processes, it could vote for a different value other than its initial value in round 1. It could even skip round 1, and vote for some value in a later round. Thus, we recognize any value that a faulty process claims as its initial value, because we may never know its "real" initial value, and all the claimed values and the real one are equally non-reliable.

With the definition, we firstly show the following lemma:

Lemma 29 *If a correct process decides v, then v must be chosen among $\mathcal{V}$, where $\mathcal{V}$ is the values carried by a quorum of round change messages based on round 0.*

Proof. According to Lemma 22, a correct process decides v only if v is from a correctly certified proposal of that round, so we only need to prove that any certified proposal must propose v that is chosen from a quorum of round change messages based on round 0. We do this via induction:

Base case: let r_1 be the smallest round number in which a correctly certified proposal exists. It implies that no correct process ever votes, remember that proposing is also considered as voting.

This proposal must be based on round 0, because there are no $f+1$ processes from any round $r > 0$ to acknowledge r. Both Algorithm 4.2 and Algorithm 4.3 require that a proposal based on 0 must have the value chosen from a quorum of round change messages based on round 0 (`pick_median`($\mathcal{V}$) in Algorithm 4.3 also returns a value among $\mathcal{V}$).

Induction assumption: assume the proposal of round r_2 is correctly certified, and all correct proposals before r_2 have only proposed values chosen from some quorum of round change messages based on round 0.

Induction step: the proposal of r_2 is either based on round 0, or some round $0 < r < r_2$. If it is based on round 0, it is the same as the base case. If it is based on another round r, the proposal of round r must also be correct, because at least one correct processes accepts r due to Property 2 of proposal certificate. According to the induction assumption, this proposal is also chosen from a quorum of round change messages based on 0. ∎

In other words, Lemma 29 ensures that the decided value must be chosen from the initial values of a quorum of processes, this implies a weak validity, namely the decision is from *some* process.

We can also achieve median validity if $n > 3f$ and Algorithm 4.3 is applied. Consider the following lemma:

Lemma 30 *Let A be an vector which contains the initial values of any $(n-f)$ processes. Then* `median`(A) *is always valid according to Definition 5 of median validity:*

Proof. Denote the actual number of faulty processes during runtime as $\tilde{f}$, so $\tilde{f} \leq f$. All other notions keep the same: the number of actual correct processes is $n_c = n - \tilde{f}$; $Sorted\_Correct$ is an array containing these n_c correct initial values and sorted in an ascending order (the index starts from 0); and $c := \lfloor \frac{n_c - 1}{2} \rfloor$, namely the index of the median of $Sorted\_Correct$.

A value v is valid according to Definition 5, if and only if $Sorted\_Correct[c-f] \leq v \leq Sorted\_Correct[c+f]$. The left part of the inequality is true if and only if all the values in $Sorted\_Correct$ starting from index 0 until index $c-f$ (the $c-f+1$ smallest values) are not greater than v. The right part is true if and only if all the values from index $c+f$ till the end (the $n-\tilde{f}-c-f$ greatest values) are not less than v. Formally, v is valid if and only if:

$$|\{v' \in Sorted\_Correct | v' \leq v\}| \geq c - f + 1 \tag{4.1}$$

$$|\{v' \in Sorted\_Correct | v' \geq v\}| \geq n - \tilde{f} - c - f \tag{4.2}$$

Note that sometimes we treat vectors as a multiset, such as $Sorted\_Correct$ and A, and all sets through this proof are multisets that allow duplicated instances. Now we prove that `median`(A) satisfies the above two inequalities. The size of A is $n-f$, according to the definition of median, there must be:

$$|\{v' \in A | v' \leq \texttt{median}(A)\}| \geq \lfloor \frac{n-f-1}{2} \rfloor + 1 \tag{4.3}$$

Among A there are at most $\tilde{f}$ faulty values, and $\tilde{f} \leq f$. If we eliminate all the faulty values from A, we have:

$$\begin{aligned} &|\{v' \in A \cap Sorted\_Correct | v' \leq \texttt{median}(A)\}| \\ &\geq \lfloor \frac{n-f-1}{2} \rfloor + 1 - \tilde{f} = \lfloor \frac{n-f-1}{2} \rfloor + 1 + f - f - \tilde{f} \\ &\geq \lfloor \frac{n-\tilde{f}-1}{2} \rfloor + 1 + \tilde{f} - f - \tilde{f} \overset{c=\lfloor \frac{n-\tilde{f}-1}{2} \rfloor}{=\!=\!=} c - f + 1 \end{aligned} \tag{4.4}$$

Because $A \cap Sorted\_Correct \subseteq Sorted\_Correct$, we can confirm:

$$|\{v' \in Sorted\_Correct | v' \leq \texttt{median}(A)\}| \geq c - f + 1 \tag{4.5}$$

This corresponds exactly equation 4.1. Similarly we can proof equation 4.2. ∎

We can now conclude the theorem regarding the validity:

Theorem 31 RATCHETA *can achieve weak validity, namely the decided value is a initial value of some process, if* `can_prove` *is implemented as in Algorithm 4.2. If* $f \leq \lfloor \frac{n-1}{3} \rfloor$ *and the values are comparable,* RATCHETA *can further achieve median validity, if* `can_prove` *is implemented as in Algorithm 4.3.*

Proof. Weak validity is proved in Lemma 29.

As discussed in Remark 2 of Chapter 2, an indispensable requirement of median validity is $f \leq \lfloor \frac{n-1}{3} \rfloor$. If this condition holds, we can require the quorum size of a proposal certificate to be $n-f$, so the decided value must be the median of $n-f$ initial values. Furthermore, Lemma 30 ensures that this value fulfills median validity. ∎

Note that $f \leq \lfloor \frac{n-1}{2} \rfloor$ is enough for consensus and weak validity, but the median validity requires $f \leq \lfloor \frac{n-1}{3} \rfloor$ according to the discussion in Remark 2.

4.5.4 Limited Memory Usage and Message Size

As discussed before, compared to most other works of hybrid fault-tolerant consensus, one important feature of RATCHETA is that it can guarantee a limited memory usage and message size. Formally, we have the following theorem:

Theorem 32 *Each process requires* $\mathcal{O}(n^2)$ *space in memory to execute* RATCHETA. *Every message has the size of* $\mathcal{O}(n^2)$.

Proof. Besides the constant-sized variables such as $vote$, r and r^{last}, the only variables with dynamic sizes are $lACK$ and $CERT^{last}$. The round-change message can contain $lACK$, and every

vote message or late-acknowledgement is accompanied by $CERT^{last}$, so we only need to prove that $lACK$ and $CERT^{last}$ have limited sizes.

The $lACK$ set contains at most $n-1$ items by the end of each round. A process may receive multiple late-acknowledgement messages from the same sender in one round. This can be due to faulty behaviors, or be caused by communication delay. We can fix the issue by only storing the highest round acknowledged by each sender. This change does not affect the correctness proved so far, especially termination in Theorem 28.

The certificate $CERT^{last}$ contains at most a quorum of round-change messages. Each round-change message has the set $lACK$ and several other fixed-sized field, so $CERT^{last}$ has the size of $\mathcal{O}(n^2)$. ∎

4.6 Optimization of Quorum Size

A quorum in RATCHETA consists of $\lceil \frac{n+1}{2} \rceil$ processes to tolerate as many as $f = \lfloor \frac{n-1}{2} \rfloor$ faulty processes. This is because we require any two quorums intersect with at least one process. If the group size n is an odd number, two quorums intersect with exactly one process. However, if n is an even number, the quorum size is not optimal. Inspired by the idea of Flexible Paxos [34], the condition that "any two quorums must intersect with at least one process" can be relaxed.

4.6.1 Revised Definition of Quorum

Recall RATCHETA algorithm, there are two places where a quorum is required. The first is in the `can_prove` function to verify a proposal certificate, and the family of all these quorums is denoted as $\mathcal{Q}_p$. The second is where a process can safely decide a value, and the family of all such quorums is $\mathcal{Q}_d$. The purpose of the quorum intersection is that when some process has decided, this information is carried into the following rounds by at least one process, and it cannot be concealed because of BiTrInc. To achieve this, it is enough to require that any quorum of $\mathcal{Q}_d$ intersects with any quorum of $\mathcal{Q}_p$:

$$\forall Q_1 \in \mathcal{Q}_d \, \forall Q_2 \in \mathcal{Q}_p : Q_1 \cap Q_2 \neq \emptyset \tag{4.6}$$

Unlike Flexible Paxos, we have to take Byzantine faults into account, so there are two further constraints:

$$\begin{aligned} \forall Q_1 \in \mathcal{Q}_d : |Q_1| > f \\ \forall Q_2 \in \mathcal{Q}_p : |Q_2| > f \end{aligned} \tag{4.7}$$

$$\begin{aligned} \forall F \subseteq \Pi(|F| = f)\, \exists Q_1 \in \mathcal{Q}_d : F \cap Q_1 = \emptyset \\ \forall F \subseteq \Pi(|F| = f)\, \exists Q_2 \in \mathcal{Q}_p : F \cap Q_2 = \emptyset \end{aligned} \tag{4.8}$$

The former (4.7) requires that each quorum contains at least one correct process, whereas the latter (4.8) requires that even if f processes are faulty and no matter which f, there must be a quorum Q_1 and a Q_2 containing only non-faulty ones. Obviously, our original definition with $\mathcal{Q}_d = \mathcal{Q}_p$ containing all combinations of $\lceil \frac{n+1}{2} \rceil$ processes can satisfy all the three requirements. We can also see that if n is an odd number and $f = \frac{n-1}{2}$, this definition is optimal because we cannot remove any subset Q from them, nor remove any process from any subset Q. However, if n is an even number and $f = \frac{n}{2} - 1$, there is a margin for improvement. For example, we can keep $\mathcal{Q}_d$ still contain all subsets of $\frac{n}{2} + 1$ processes, while $\mathcal{Q}_p$ consists of only subsets of $\frac{n}{2}$ processes. If $n = 4$, then $\mathcal{Q}_d = \{\{p_1, p_2, p_3\}, \{p_2, p_3, p_4\}, \{p_1, p_3, p_4\}, \{p_1, p_2, p_4\}\}$, and $\mathcal{Q}_p = \{\{p_1, p_2\}, \{p_2, p_3\}, \{p_3, p_4\}, \{p_1, p_4\}, \{p_1, p_3\}, \{p_2, p_4\}\}$. Even better, we can make both $\mathcal{Q}_d$ and $\mathcal{Q}_p$ containing only subsets of $\frac{n}{2}$ processes: $\mathcal{Q}_d = \{\{p_1, p_3\}, \{p_2, p_4\}, \{p_1, p_4\}, \{p_2, p_3\}\}$ and $\mathcal{Q}_p = \{\{p_1, p_2\}, \{p_3, p_4\}\}$. Even if no quorum reaches the majority in this case, consensus can still be ensured. Figure 4.9 depicts an example of this case and gives a rough idea how this works. In round 1, p_1 and p_3 have already decided because they themselves can comprise a quorum to decide. In the next round, p_2 is the coordinator and has received round-change messages from itself and p_4. However, this cannot let p_2 propose, because $\{p_2, p_4\}$ is not a quorum of $\mathcal{Q}_p$. We can see that any quorum in $\mathcal{Q}_p$ must contain either p_1 or p_3, which has voted in round 1. This information cannot be concealed because of BiTrInc, so any new proposal cannot be based on a round earlier than 1.

4.6.2 Revised Proof of Agreement

As the definition of quorum is modified, the correctness proofs must be revised accordingly.

Proof (revised for agreement). We adopt the same tactic by proving that:

- no two correct processes decide different values in the same round (Lemma 23); and
- if a correct process decides v in round r, no valid proposal for another value exists in any later rounds (Lemma 24 and Lemma 25).

The correctness of Lemma 23 is based on BiTrInc and that any quorum contains at least one correct process, so it is not impacted under the new definition of the quorum.

Lemma 24 states that once a correct process decides in round r, then in any later rounds, there exists no valid proposal based on round $r^- < r$. This relies on BiTrInc and any quorum $Q_1 \in \mathcal{Q}_d$ must intersect with any $Q_2 \in \mathcal{Q}_p$. More specifically, if any quorum $Q_1 \in \mathcal{Q}_d$ voted in

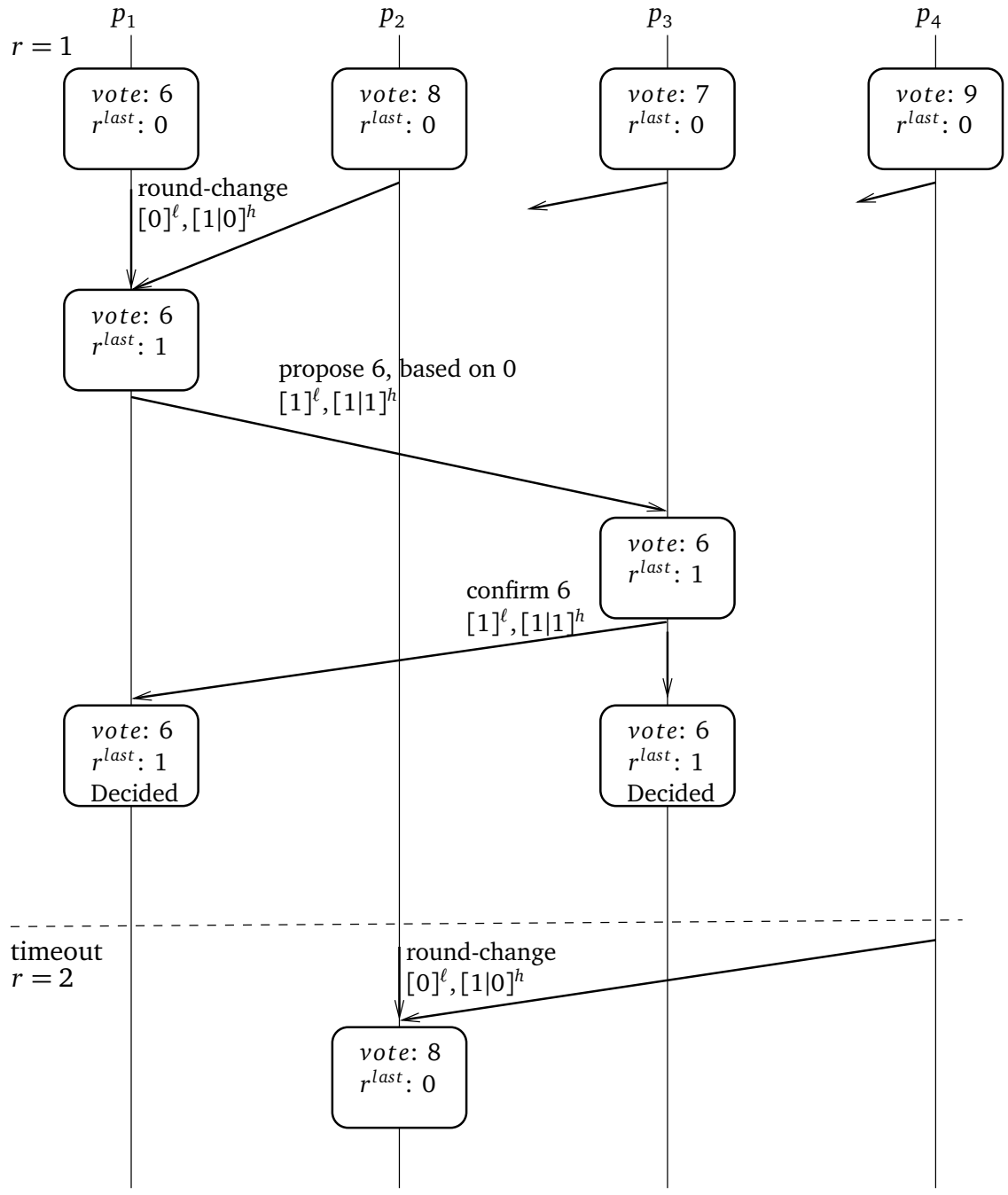


Figure 4.9: An example of the new definition of quorum ($n = 4, f = 1$, fault-free). Here $\mathcal{Q}_p = \{\{p_1, p_2\}, \{p_3, p_4\}\}$ and $\mathcal{Q}_d = \{\{p_1, p_3\}, \{p_2, p_4\}, \{p_1, p_4\}, \{p_2, p_3\}\}$. In round 1, p_1 can propose with the help of p_2, and p_1 and p_3 can decide with the help of each other. In round 2, p_2 cannot propose with the help of p_4 alone, because they do not constitute a quorum of $\mathcal{Q}_p$.

round r, they must have set their counters ℓ to r, so in a later round, there is no quorum $Q_2 \in \mathcal{Q}_p$, in which all processes are from r^- or other earlier rounds.

Lemma 25 ensures that once a correct process decides v in round r, all later rounds can only propose v as well. Its correctness relies on the property of the proposal certificate that at least $f+1$ round-change messages must be included, in which at least one is from a correct process. Because $\forall Q_2 \in \mathcal{Q}_p : |Q_2| > f$, the correctness still holds.

In conclusion, agreement can be guaranteed in the revised algorithm. ∎

4.6.3 Revised Proof of Termination

Proof (revised for termination). Lemma 27 states that there must be a round r after GST, and its next round $r+1$ has a correct coordinator. This is independent of the quorum definition. According to Equation 4.8, the set of all correct processes must be the superset of some $Q_1 \in \mathcal{Q}_d$ and $Q_2 \in \mathcal{Q}_p$. Using the same proof technique as in the original proof, we can conclude that all correct processes can terminate in round $r+1$. ∎

4.6.4 Revised Proof of Validity and Limited Memory Usage / Message Size

Proof (revised for validity). The weak validity version of `can_prove` function guarantees that the decided value is chosen from a set of round change messages based on round 0, while the median validity version guarantees that the decision is the median of $(n-f)$ initial values ($n > 3f$ is still necessary). They are independent of the definition of quorums. Thus, the validity still holds. ∎

Proof (revised for limited memory usage and message size). The changed size of a quorum, especially the round change quorum $\mathcal{Q}_p$, may influence the size of a proposal certificate. Nevertheless, $|\mathcal{Q}_p|$ cannot exceed n, so the space complexity of each process and each message is $\mathcal{O}(n^2)$. ∎

4.7 Other Optimization

Besides the definition of the quorum (especially in a even-number-sized group), there are several other possibilities of optimization.

4.7.1 Decision Forwarding

Similar to TRUSTED BEN-OR 3.4.3, we can use the decision forwarding technique to accelerate termination once some correct process has decided. Instead of keeping changing rounds, the decided process can immediately stop the original algorithm but only repeatedly broadcast its decision. The decision does not require a specific counter authentication, but has to include the quorum of vote messages that makes the process decide. This modification will not break the correctness. Recall that when we prove the agreement, we actually prove that if a quorum $Q \in \mathcal{Q}_d$

has voted the same value in some round, no correct process will decide a different value. As for the termination, validity and limited space complexity, the correctness is trivial.

4.7.2 Skip the Round Change in the First Round (Weak Validity)

Weak validity requires that the decided value must be the initial value of some process, regardless of which one, faulty or not. If this is the case, there is no need to exchange the initial values in the first round ($r = 1$). Instead, the coordinator of the first round directly proposes its own value without any certificate. The non-coordinator processes can vote for the proposal as long as it has the correct BiTrInc authentication. This only applies for round $r = 1$. All the later rounds still need the round change phase, and the proposals must be correctly certified.

4.7.3 Stubborn Broadcasting

RATCHETA does not assume reliable communication. Every `broadcast`(m) in the algorithm is simply send a message m out, without any acknowledgement or assurance that the message is delivered. In a real application scenario, the message can indeed get lost. To overcome the possible message loss and increase the possibility that a message is delivered by every one, a process can repeatedly broadcast its last sent message before it enters the next phase.

However, too frequent broadcasting may saturate the network and cause distributed denial-of-service (DDoS) attack. How to choose a proper stubborn broadcasting frequency and to prevent potential malicious DDoS attack depends on the specific application scenarios.

4.8 Evaluation

We have conducted experiments on the same testbed as in the previous chapter to evaluate the fault-tolerance and performance of RATCHETA.

4.8.1 Testbed and Methodology

The testbed consists of ten nodes of Raspberry Pi 3 (model B), connected in a wireless ad-hoc network. The hardware specification and settings are the same as in Section 3.5 and we briefly recapitulate the important experiment settings here. All nodes are distributed in different rooms on the same floor in our office building. The trusted subsystem to contain BiTrInc is built on top of OP-TEE [67] based on ARM TrustZone [5]. The RATCHETA algorithm is implemented in C++, while BiTrInc running inside the trusted subsystem is written in C because OP-TEE lacks C++ support, as mentioned before. BiTrInc uses the SHA-256-based HMAC algorithm for message authentication. The secret key is correctly distributed in advance. All nodes share the same secret key, because the trusted subsystem can ensure that even a malicious host cannot access the secret key.

When the RATCHETA algorithm needs to authenticate or verify a message via BiTrInc, OP-TEE performs world switch from the normal world to the secure world to do the calculations. After BiTrInc finishes its job, the results are written to a memory chunk shared by both the normal and secure world, and the execution is switched back to the normal world.

The network condition is also described in Chapter 3.5. The minimal, median and maximum of the round trip time of an ICMP ping message is 5.6 ms, 12.5 ms and 1356.7 ms respectively. The UDP link between two nodes has a jitter of 139.9 ms as reported by the *iperf3* tool, which stands for a high variance of the communication delay. The packet loss rate reported by *iperf3* is as high as 24%.

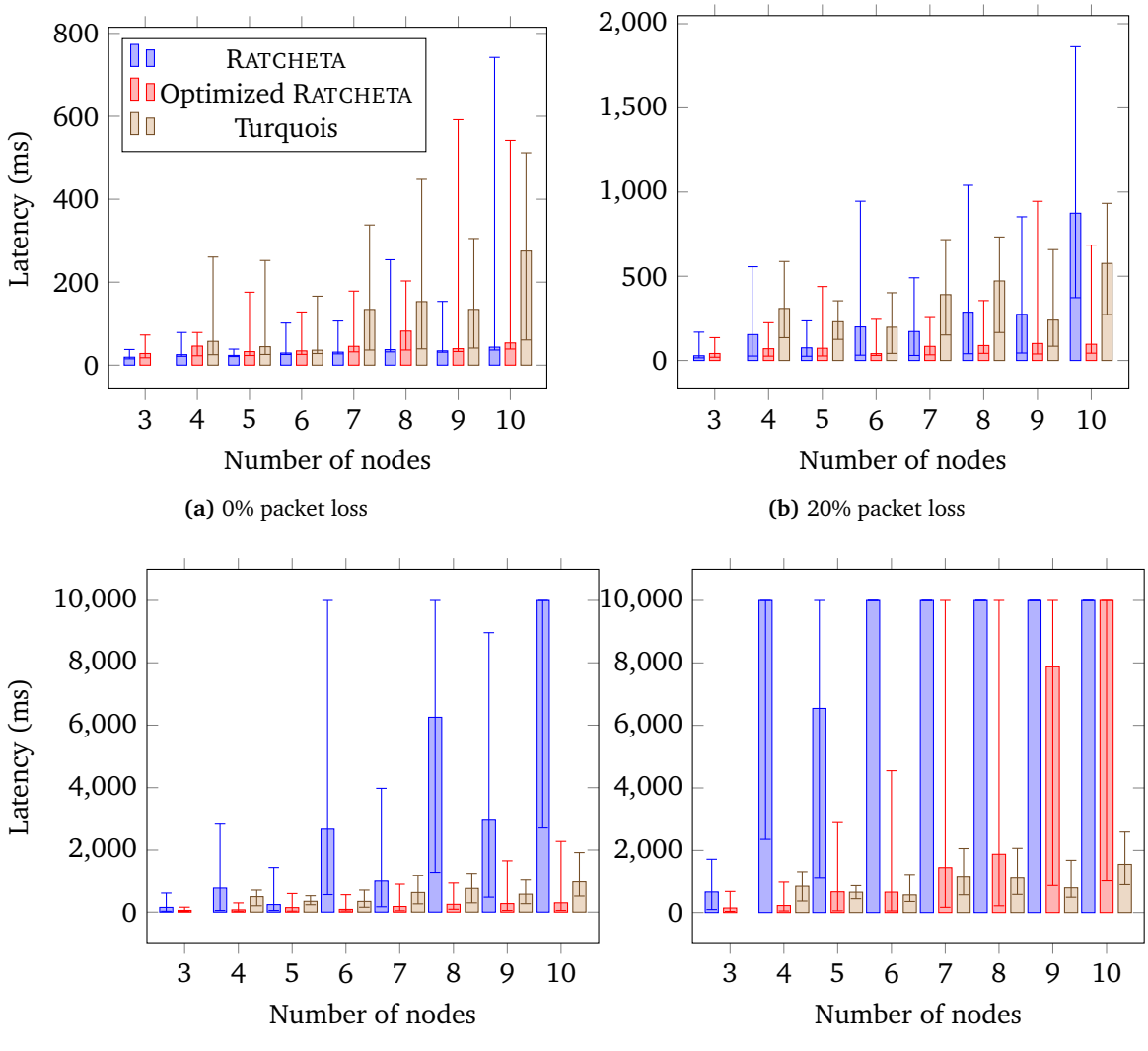
The timeout of a round change is initialized as 50 ms. In every further round, this timeout is increased by 10 ms.

The methodology is similar to what we used to evaluate TRUSTED BEN-OR. The experiment consists of two scenarios: fault-free scenario and Byzantine fault scenario. In both scenarios, message omissions are injected by artificially dropping packets with a configurable packet loss rate at the recipient side. We evaluate the performance with the latency between the start of the algorithm and the last node deciding. Every process is assigned a different initial value to prove that RATCHETA is a multi-value consensus algorithm. The results of the original RATCHETA and the optimized version with all methods described in Section 4.6 and 4.7 are evaluated. The same as in the evaluation of TRUSTED BEN-OR, we choose Turquois as a baseline for comparison. Although there are other hybrid fault-tolerant consensus algorithms [71, 37, 6], we do not choose them for comparison because they are all designed for state machine replications and contain a lot of unnecessary features in our application use cases. The test cases are categorized according to the group size and the injected packet loss rate. In each test case, we run each algorithm 100 times. Furthermore, we manually set a deadline to 10 seconds to force the algorithm to terminate, to avoid a too long waiting time under a high packet loss rate. This deadline is just for evaluation purpose and does not belong to the algorithm specification.

4.8.2 Experiment with Non-Faulty Processes and Omission Faults

The first experiment does not consider any Byzantine faults, namely all nodes are correct, but message omissions can occur. The test cases cover group sizes from 3 to 10, and packet loss rate 0%, 20%, 40% and 60%. The results are shown in Figure 4.10. Because we set a deadline of 10 seconds, the latency reaching 10 s can be regarded as non-terminated before the deadline.

From Figure 4.10a we can see that under a good network condition without extra omissions injected, the median termination latency of RATCHETA— both non-optimized and optimized versions — is below 100 ms in every test case. Moreover, RATCHETA shows a good scalability, because the median latency does not increases significantly as the group size grows. This is because that RATCHETA can terminate in only one round regardless of the group size, as long as there are no Byzantine nodes and the network is perfect. In our testbed, it takes about 50 ms



(a) 0% packet loss

(b) 20% packet loss

(c) 40% packet loss

(d) 60% packet loss

Figure 4.10: Median of latency of the original and optimized RATCHETA in fault-free case in comparison to Turquois. The error bar represents the 9th and 91st percentile. The legend in sub-figure (a) applies for all sub-figures. Note that 10,000 ms means that the algorithm fails to terminate within 10,000 ms.

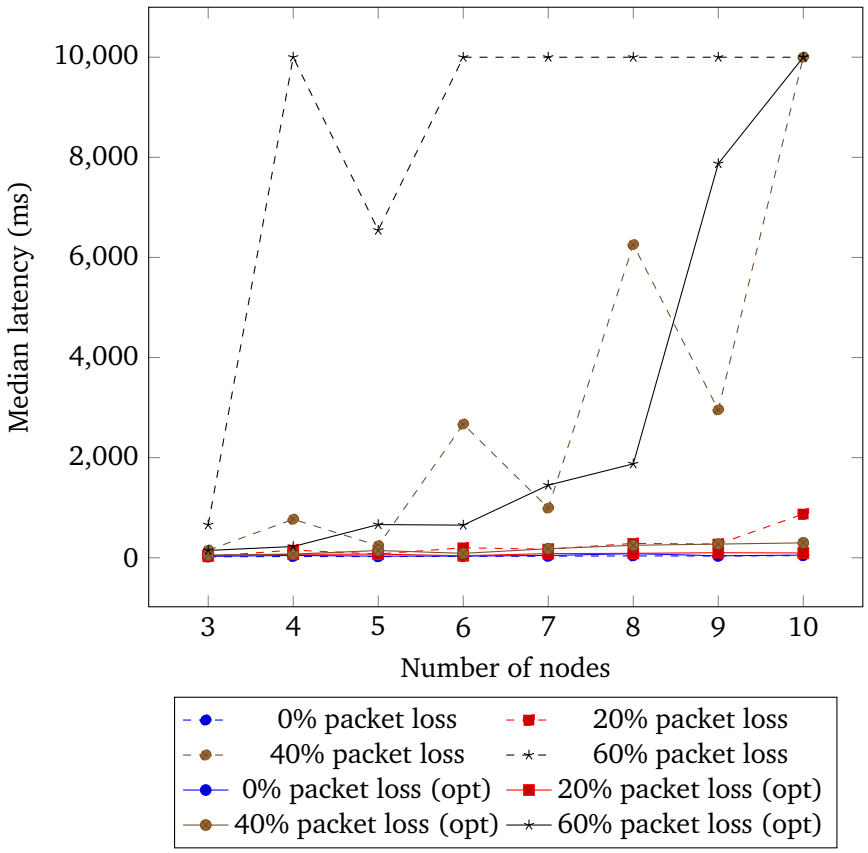


Figure 4.11: Median of latency of RATCHETA in fault-free case. The (opt) in legend indicates the optimization is enabled. Note that 10,000 ms means that the algorithm fails to terminate within 10,000 ms.

to terminate within one round. Only in seldom cases, it takes two or more rounds to terminate due to real packet losses (not the artificially injected ones), for instance the median latency with $n = 8$ in the optimized version. In contrast, a randomized algorithm like Turquois has to wait until a lucky round when all participants randomly get the specific value. The more participants in the group, the more rounds are expected, which leads to a poor scalability of the randomized consensus. Nevertheless, the variance of RATCHETA also becomes large when the group size grows. The reason is that the network is far from perfect and omissions exist even without fault injection.

Another observation in the case of 0% omission faults is that the optimized RATCHETA seems to be even worse than the non-optimized version. The reason lies in the fact already discussed above: in the best case, RATCHETA can terminate in only one round regardless of the optimizations. Thus, the optimizations cannot bring any benefit in this case.

The remaining three sub-figures show that if the network condition becomes severe where the packet loss rate is above 20%, the optimization has a significant improvement in efficiency. When omission faults are injected, the results show that even with up to 40% omission faults injected, the optimized RATCHETA still performs well and also scales well, but the performance of the non-

optimized RATCHETA quickly degrades. The reason that the non-optimized RATCHETA performs poorly with increased omission faults is that RATCHETA relies on timeouts to change rounds. The timeout interval must be increased from round to round due to the partially synchronous system model. The side effect of the increased timeout is that if the algorithm cannot terminate within the first several rounds, each round-change will take longer and longer. Under a severe network condition with 60% artificial packet loss, the non-optimized RATCHETA can hardly work, and the optimized version may also fail to terminate within 10 seconds with $n \geq 7$, but still works well in a small group of $n = 3$ or 4.

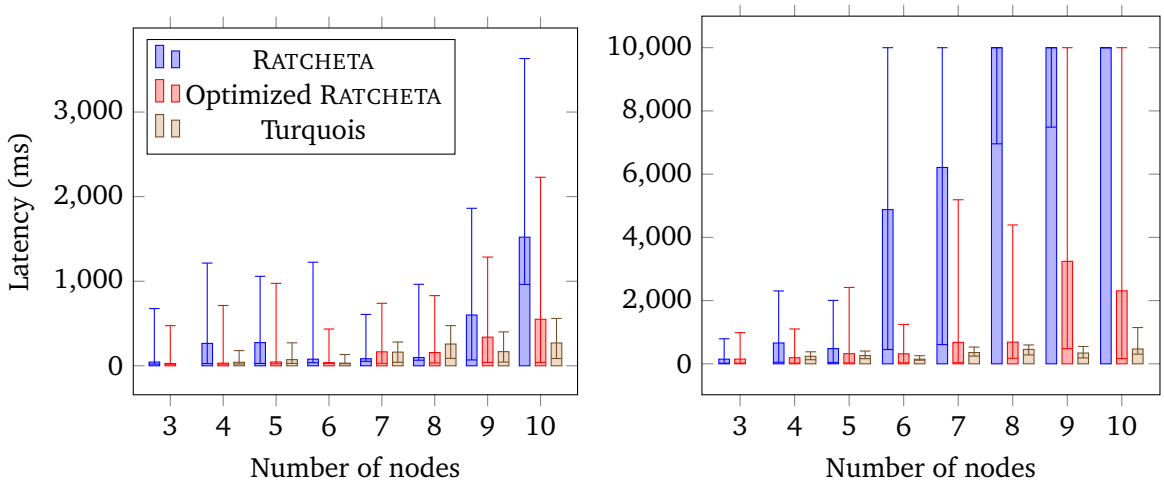
In the end, we plot the median latencies of each test case together in the same line graph of Figure 4.11. The graph confirms the good performance and scalability of the optimized RATCHETA, when the packet loss rate does not exceed 40%. It also clearly shows that the optimization can bring significant benefit compared to the non-optimized one.

4.8.3 Experiment with Byzantine Processes and Omission Faults

In the second experiment, Byzantine faults are injected. Because RATCHETA can tolerate up to $\lfloor \frac{n-1}{2} \rfloor$ faulty nodes, we let $\lfloor \frac{n-1}{2} \rfloor$ nodes in the group be Byzantine, which means that only weak validity is tested here. Specifically, Byzantine nodes will not send the value that it is supposed to send, but will send an arbitrary faulty value in their messages. The results of latency are shown in Figure 4.12. With Byzantine nodes included, RATCHETA (optimized) does not perform as well as in the previous experiment. The latency of $n = 10$ and 0% packet loss rate is already above 500 ms. Starting from 40% packet loss rate and $n = 7$, most cases fail to terminate within the 10 seconds' deadline.

The median latencies of the optimized RATCHETA are collected and plotted in Figure 4.13. The non-optimized version is not included. It shows that in a very small group with $n = 3$ or 4, the latency is still relatively small with 20% packet loss. As the group size increases, the latency grows and sometimes even with a sudden jump. For example in the 20% packet loss case, the latency of $n = 9$ is over three times as much as $n = 8$. In the 40% packet loss case, a group of $n = 6$ has a median latency of 1086 ms, while in a group of 7, the median latency is already beyond the 10 seconds' deadline, which increases over 10 fold.

The results show that the latency of RATCHETA is sensible to Byzantine faults. The reason is that there is a unique coordinator in each round and round change is driven by timeout. If the coordinator of the current round is faulty, a correct process can do nothing but wait for the next round, even if it has collected a quorum of correct messages. Moreover, the round-change timeout increases in every new round. Such a design has two considerations. Firstly, we do not know about the clock drifts among processes, so a round should be long enough for all processes to stay in the same round. Secondly, we do not know the possible communication delay in a partially synchronous system, so a round should also be long enough for every message to get delivered. As a result of this timeout-increasing, if the algorithm fails to terminate in the first few rounds, either



(a) 0% packet loss

(b) 20% packet loss

(c) 40% packet loss

(d) 60% packet loss

Figure 4.12: Median of latency of the original and optimized RATCHETA with Byzantine nodes in comparison to Turquois. The error bar represents the 9th and 91st percentile. The legend in sub-figure (a) applies for all sub-figures. Note that 10,000 ms means that the algorithm fails to terminate within 10,000 ms.

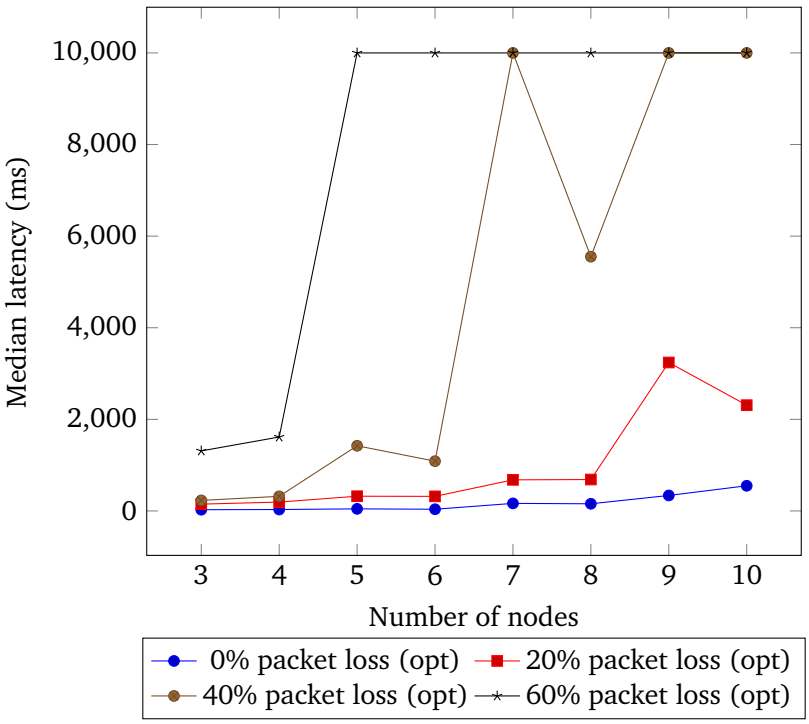


Figure 4.13: Median of latency of the optimized RATCHETA with Byzantine nodes.The (opt) in legend indicates the optimization is enabled. Note that 10,000 ms means that the algorithm fails to terminate within 10,000 ms.

because of a faulty coordinator or message omission, the timeout will easily become very large. To avoid this issue, a fixed timeout can be chosen for round change, if the system designer has a strong confidence about the maximum clock drifts and the maximum transmission delay (after the system becomes synchronous).

4.9 Related Work

Multi-value BFT consensus: most of the multi-value consensus algorithms are based on a convergence approach [77, 56, 75], in which the values of each process converge (e. g. to their average) via iterative value exchanging and updating. This approach is similar to the approximate consensus [21], where the processes can decide different values, but must be close enough to each other. Both the convergent and approximate consensus are widely used in sensor data fusion [4], clock synchronization [62], and control theory in time-varying systems [60]. But these approaches commonly only consider — if ever — a subset of Byzantine faults such as the noise or incorrectness of data input, while the equivocation is not covered. LeBlanc et al. [41] explicitly tackled the equivocation attack in multi-agent systems. Nevertheless, a synchronous communication model is assumed in their work, which is less realistic for our target applications. Moreover, all the convergent consensus algorithms require a comparable value

space such as numeric values—the same requirement applies for median validity. If the values are not comparable, it is hardly to define how to converge different values. Agreeing on a search plan in our previous discussion (Section 4.2) is such an example where the convergent consensus cannot work. The reason is obvious: if a node has its own plan and receives several different plans from others, how to calculate a new plan is not trivial unless we explicitly define how to converge all these plans.

Correia et al. discussed how to transform randomized binary consensus to multi-value consensus [17]. But in their algorithm, the processes can decide on a default value $\perp$ that is not proposed by anyone. This assumption can be hardly used in applications where a concretely agreed value is required in the end, e. g. in our life rescue scenario.

Hybrid fault-tolerant consensus. Correia et al. [18] show how to transform a crash consensus to Byzantine consensus in asynchronous systems with hybrid fault model. Their work is mainly from a theoretical point of view, and provides methodology of such transformations on an abstract level. We also introduced several practical works applying the hybrid fault model [16, 43, 71, 37, 6] in section 4.2.2. We discussed their limits, including that 1) most of them [16, 43, 71, 37] cannot guarantee a bounded memory usage and 2) almost all of them are designed for state machine replication and are not suitable to be directly applied on wireless embedded systems. For instance they normally assume a low-level protocol such as TCP to provide a reliable message delivery.

XFT [47] assumes a different hybrid fault model tailored for wide area networks, in which the communication between the correct nodes is reliable for most of the time, but this assumption applies less likely to an open wireless network.

4.10 Conclusion

In this chapter, we present RATCHETA, a Byzantine fault tolerant consensus algorithm designed for cooperative wireless embedded systems. RATCHETA can tolerate as many as $\lfloor \frac{n-1}{2} \rfloor$ Byzantine processes among n processes, which is the optimum achievable result in a partially synchronous system. We assume a hybrid fault model, in which a special trusted subsystem is used to prevent equivocation. The trusted subsystem contains a counter-based authentication service named BiTrInc. By using two monotonic counters in parallel, BiTrInc can effectively prevent equivocation attacks. Moreover, it solves the problem of potentially unbounded memory usage and message size that is inherent to several other works. In our prototype, the trusted subsystem is protected by the ARM TrustZone that is available on many embedded devices. RATCHETA uses UDP multicast to reduce the communication overhead and does not rely on any reliable messaging mechanisms of the low-level protocols. The experimental results show that RATCHETA works well when Byzantine faults are present and the network condition is poor. In a majority of experiment cases on the real hardware testbed with potential packet losses, a group with no more than 10 nodes can achieve

consensus under 100ms, even if extra 20% packet losses are injected. With $\lfloor \frac{n-1}{2} \rfloor$ nodes being Byzantine, a group of 10 nodes can also terminate within 550 ms without extra omission faults injected.

5

Consensus for Autonomous Maneuver Coordination

In this chapter, we are going to introduce an application scenario of consensus that is used on autonomous vehicles, namely, the maneuver coordination via vehicle-to-vehicle (V2V) communication.

5.1 Motivation

Autonomous vehicles, compared to human drivers, have the advantage to obtain and process more information and react more quickly and precisely, leading to increased safety, road capacity and fuel efficiency [46]. Such information can be provided not only by on-board sensors, but also by other vehicles via V2V communication. This way, the connected vehicles are able to exchange instant and precise driving information with each other to improve their driving safety and efficiency. For example, with the V2V communication, vehicles can drive at closer distances to form a *platoon* [12, 73, 60, 45, 20]. The already standardized Cooperative Awareness (CA) service [25] also enables autonomous vehicles to get informed about the driving status of the surrounding connected vehicles. Via the Collective Perception service [27, 29, 33], a vehicle can also share with others about the information of the non-connected objects it perceives. With the help of all the shared information, vehicles have a better knowledge about the environment thus can avoid dangerous driving behaviors that would potentially cause accidents.

The aforementioned examples utilize V2V communication to exchange information for a cooperative and coordinated driving. Other situations where coordination is useful are lane-changing and crossing uncontrolled intersections. By default, the coordinated driving behavior

in these situations is defined by the right-of-way rules, such as “priority to the right” in most of the countries of right-hand traffic. However, in certain cases this is not the optimal solution regarding the traffic efficiency. By leveraging V2V communication, vehicles can have a global view about the driving plan of everyone, and can negotiate to reach an agreement, so that they can cooperate to make the best use of traffic resources, alleviate the traffic pressure and reduce energy consumption.

This maneuver coordination can be viewed as a consensus problem, where a group of vehicles agree on a conflict-free maneuver plan. Therefore, a distributed consensus algorithm can be applied to achieve the goal. The fault-tolerance also has to be considered in this safety-critical application. Because this specific use case scenario has its own requirement, as we will see later, we cannot directly take an off-the-shelf consensus algorithm. Lehmann et al. [42] describe a generic approach of maneuver coordination based on trajectory exchange. In this chapter, we extend this idea with the fault-tolerant consensus. The features of our work include:

- We have designed a concrete coordination protocol for the Maneuver Coordination service. This fills the missing part in the original paper [42] where the Maneuver Coordination service is proposed. The original work only introduces the framework of the maneuver coordination, but they do not describe how consensus can be achieved in this framework.
- The protocol is actually a special consensus algorithm, as in most cases it can achieve consensus and prevent divergence, i. e. the involved vehicles end up with decisions contradictory to each other.
- The fault-tolerance of the protocol as well as the impact of communication failures is discussed. We show that in case the vehicles fail to reach consensus, the impact is restricted to a relatively low level. The safety is not violated.

5.2 Related Work

An important application of the vehicle cooperation is platooning, where a group of vehicles drive consistently to reduce the distance between them and to improve the road capacity. A platoon can be implemented via the Adaptive Cruise Control (ACC) [48], in which the control information comes purely from on-board sensors. Later, the Cooperative Adaptive Cruise Control (CACC) was proposed, which utilizes vehicular communication in addition to the on-board sensors. Researches show that with vehicular communication, a platoon can achieve higher road capacity [68] and reduced energy consumption [64]. Most previous works assume that a platoon with a fixed number of vehicles already exists [72, 20]. Unlike those, Li et al. [44] have designed an algorithm to let individual vehicles to form platoons spontaneously.

In a platoon, all vehicles cooperate to achieve the same goal. Besides platooning, there are also other application scenarios where vehicles have potential conflicting plans and they can

utilize V2V communication to coordinate their maneuvers and resolve conflicts. Such scenarios include lane-changing and merging, entering intersections or roundabouts, just to name a few. Previous research on cooperative merging maneuvers by Kazerooni and Ploeg [38] have proposed a protocol to enable a vehicle to merge into a platoon in case of an oncoming lane reduction. In their system, the platooning vehicles increase their time gap towards their respective predecessor, creating a gap for the merging vehicle. Their approach assumes an already established cooperation between the platoon members, but does not cover the generalized merging maneuver without preexisting platoons. Wang et al. proposed an on-ramp merging scheme for highways [74]. They utilize Road Side Units (RSU) to determine the sequential order of vehicles on the ramp and the highway at the time of the upcoming merge. The RSU then assigns each vehicle on the highway or about to enter the highway a predecessor vehicle. Based on the assigned order, a platoon is formed and all vehicles adjust their own speed to leave a gap for merging. This approach requires that all vehicles are able to communicate via vehicle-to-everything (V2X) and relies on the presence of RSUs at on-ramps.

The above-mentioned works focus on a specific scenario, i. e. lane-merging. Recently Lehmann et al. [42] propose a generic approach for maneuver coordination, which is independent of specific traffic scenarios. Nevertheless, the authors discussed less on the concrete coordination protocol, but focused more on the conceptual design. In this work, we build a coordination protocol based on their idea, and explain how to prevent contradictory decisions in the coordination and analyze the impact of communication failures.

5.3 Preliminary: Maneuver Coordination Service

Before we illustrate the coordination protocol, we give a brief overview of the Maneuver Coordination service. For more details, we encourage the readers to refer to the work of Lehmann et al. [42]. The purpose of the Maneuver Coordination service is to enable autonomous vehicles to form a spontaneous group and then drive in a coordinated manner. It is designed for the *European Telecommunications Standards Institute (ETSI) Intelligent Transport System G5 (ITS-G5)* standard [26], so it relies on several other facilities of ITS-G5, including the Position and Time Management that is still under development [30].

To achieve coordinated driving, the Maneuver Coordination service consists of three phases:

Detection phase The autonomous vehicles can perceive the environment and plan their maneuver behaviors of the near future. Every vehicle periodically broadcasts a message including its *planned trajectory* (similar to the Cooperative Awareness Message of ITS-G5 [25]). The trajectory is a spatial-temporal description of the vehicle's movement represented as Frenét frames [76], which is a sequence of data sets containing the time, the arc length of the lane from its

beginning, as well as the perpendicular offset in the lane. In simple words, the trajectory indicates how a vehicle will move in the near future. This can be realized with existing techinque [36].

The periodic trajectory broadcasting has two purposes. Firstly, the vehicle demonstrates its identity, showing that it is equipped with the Maneuver Coordination service and is able to communicate with others. If later a coordination is necessary, the vehicles know to whom they are talking to. Secondly, it allows the vehicles to get informed about others' future movement, so they can adjust their own maneuver accordingly. For example, if one vehicle X sees its planned trajectory has a conflict to Y's, and X does not have the priority (right-of-way) in this situation, then X can change its plan in advance. Here the assumption is that even without the communication, autonomous vehicles should be able to prevent the catastrophe, but the reactions could be rougher, such as a sudden break or lane-changing.

Negotiation phase As discussed above, if a vehicle sees a conflicting trajectory from another vehicle with a higher priority, it has to change its originally planned trajectory, such as to slow down. This easiest solution does not require any further communication, but is not preferred by the low-priority vehicle. Moreover, it is not always the optimal solution with respect to the overall traffic efficiency. Alternatively, the low-priority vehicle can try to ask for a higher priority, by sending another *requested trajectory*. The requested trajectory is preferred by the sender, but can be conflicting with one or more planned trajectories of other vehicles and not correspondent with the right-of-way rules. The sender of the requested trajectory is called the *requesting vehicle*. When the affected vehicles, denoted as the *accepting vehicles*, receive the requested trajectory, they will assess: 1) if they are willing to accept, according to their local policies; 2) if they are able to execute a new trajectory to adapt to the request. If both answers are yes, every accepting vehicle will update its planned trajectory accordingly.

Execution phase After the negotiation, every vehicle will adopt its planned trajectory accordingly.

5.4 Coordination Protocol of Negotiation

Among all the three phases of the Maneuver Coordination service described in the previous section, we focus on the negotiation phase and present the proposed coordination protocol in detail.

5.4.1 Consensus in Maneuver Coordination

In the specific application scenario of maneuver coordination, the underlying problem is a distributed consensus problem, but it is slightly different to our previous consensus problems. Each vehicle may propose one or several trajectories beside its planned trajectory, indicating how

it is willing to change its maneuver if necessary. In the end, all involved vehicles must agree on a global maneuver plan. The global plan assigns each involved vehicle V_i a trajectory denoted as T_i. The negotiation should achieve:

- **Agreement**: all involved vehicles decide the same global plan.
- **Real-time termination**: all involved vehicles must decide before a given deadline.
- **Validity 1**: each vehicle V_i must be willing to perform the trajectory T_i in the plan.
- **Validity 2**: there is no conflicting trajectories in the global plan.

Note that there are several differences compared to the consensus problems we have discussed so far. Firstly, the termination property here has a real-time requirement. This is necessary for the sake of safety in the driving situation. It can be easily understood, as the planned trajectory has a time limit, e. g. a vehicle plans to "decelerate to 20 km/h in 5 seconds". It would be pointless if a vehicle make a decision later than the planned time.

Secondly, the vehicles agree on a vector of values (trajectories) instead of a single value. This is similar to interactive consistency [58]. However, in interactive consistency, each process only proposes a single value, otherwise it is a faulty process. In our case, each vehicle is allowed to propose more than one trajectories.

Thirdly, the Validity 1 is similar to the weak validity as introduced in Chapter 2, as the decided global should only include trajectories that are proposed by the vehicles. Moreover, there is one more validity requirement as stated in Validity 2, namely the global plan should be conflict-free. This is necessary because different vehicles may propose conflicting trajectories simultaneously. The reason is that a vehicle can only make a proposal based only on its current perception and already received planned trajectories from others, but cannot predict how others will change their plans. Therefore, such potential conflicts should be considered and prevented in the global plan.

Fourthly, the negotiation group is formed spontaneously, so we cannot assume that all vehicles share a common knowledge about the group members when they start the negotiation. Each one may only be aware of a subset of others via the information gathered in the detection phase.

To simplify the problem, we do not consider malicious behaviors. Namely, no vehicles will actively work against the algorithm specification to prevent others from reaching agreement. We do consider communication failures. Messages may get lost or arbitrarily delayed thus not successfully delivered by recipients. Although no malicious behaviors are considered, the situation does not become trivial and sometimes is still as complex as a Byzantine case. For instance, a vehicle is allowed to propose more than one trajectory. What is more, if no restrictions are set, a requesting vehicle can simultaneously asking priority from different accepting vehicles, while an accepting vehicle can simultaneously handle multiple requests. Theses situations are just like the equivocation attacks in BFT consensus, although not because of malicious intent.

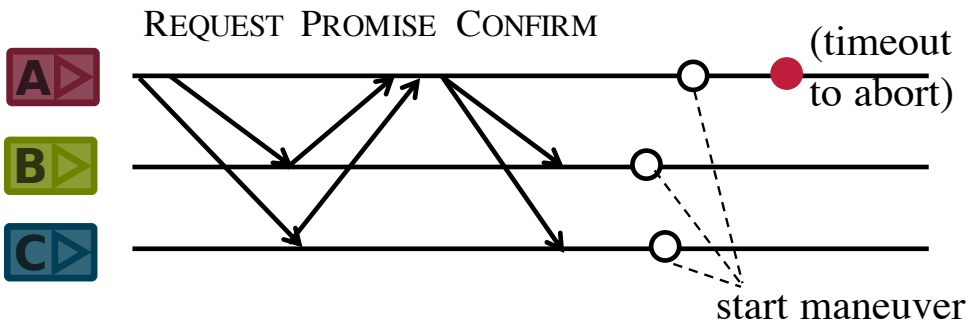


Figure 5.1: Communication pattern of the trajectory negotiation.

Apparently, it is impossible to guarantee both agreement and real-time termination under such an unreliable and asynchronous network. Unlike other algorithms we discussed so far, we allow a vehicle to abort negotiation at the given deadline. A vehicle aborts means that it decides to keep its previously planned trajectory unchanged. This will cause disagreement between vehicles, we will discuss how to tackle such disagreements later.

5.4.2 Communication Pattern

The communication pattern of the negotiation phase is shown in Figure 5.1. The negotiation is always initiated by a requesting vehicle (denoted as A here). Assume that A has received the trajectories from some vehicles around it in the detection phase. For the other vehicles that are also seen by A (via its other sensors) but A has not received their trajectories, A identifies them as "unable to communicate". The requesting vehicle A starts the negotiation if all the following conditions are satisfied:

- A's desired trajectory is conflicting with one or more accepting vehicles (here B and C);
- A has a lower priority compared to the accepting vehicles;
- No other vehicles, which are unable to communicate, are hindering A's desired trajectory.

The negotiation includes three steps:

REQUEST the requesting vehicle sends its desired trajectory to the corresponding accepting vehicles in a REQUEST message. Under some circumstances, a requesting vehicle can have multiple optional trajectories involving different accepting vehicles to fulfill its purpose. Instead of sending several REQUESTs individually, multiple REQUESTs can be batched into a single message for broadcasting to increase the communication efficiency.

PROMISE when an accepting vehicle receives the REQUEST and decides to accept it, it calculates one or more new trajectories of its own, which can facilitate the requested trajectory. The new trajectories are assembled to a PROMISE message and sent back. By doing this, it promises that it will follow one of these trajectories, as long as the requesting vehicle replies with a confirmation to pick one within a certain time.

Because this is a preliminary work, we have the simplified assumption that any promised trajectory must be the one that the accepting vehicle can freely adopt under the current driving scenario and does not require a higher priority. Examples of the allowed maneuvers include to decelerate (even if there is a following car), to change to a free lane, to accelerate if there is enough space in front, etc. This means the cascaded maneuver, in which an accepting vehicle initiates a new request to fulfill a received request, is not allowed.

CONFIRM the requesting vehicle broadcasts a CONFIRM after it can construct a global plan by selecting one trajectory from each PROMISE such that:

- its requested trajectory is not hindered anymore;
- all promised trajectories are conflict-free with each other.

The CONFIRM includes all selected trajectories of all involved vehicles, including the requesting vehicle. When the corresponding accepting vehicles receive the CONFIRM and can verify its validity, they will decide this plan and start the maneuver accordingly.

After sending the CONFIRM, the requesting vehicle sets a timeout and waits for the corresponding accepting vehicles to change the driving maneuver. It monitors whether they are driving exactly as the global plan. If so, the negotiation is successful and the requesting vehicle can also drive as it requested. But if the CONFIRM is not successfully delivered, the requesting vehicle can notice this and will also abort from the negotiation and fall back to the default behavior according to the right-of-way rule.

Besides the above mentioned steps, there are several practical optimizations. Firstly, to prevent saturating the communication channel, all the above-mentioned messages are periodically broadcast at a constant frequency, together with the planned trajectory (Section 5.3). Secondly, to increase the probability of the successful message delivery considering the possible packet losses, each vehicle will stubbornly re-transmit the last message, if it is unable to start the next step.

5.4.3 Reach Agreement to Prevent Divergence

The coordination protocol can also prevent the potential risk where different vehicles decide on contradictory plans in the end, which is called a *divergence*.

To understand how divergences can happen, we can consider the following scenarios.

Scenario 1 A single request involves more than two vehicles to change their trajectories. Because they cannot predict how others will change, they may end up with a contradictory global plan. Sometimes the trajectories of the accepting vehicles are directly conflicting with each other, leading to a safety issue. Another possibility is that the maneuver change of an accepting vehicle becomes pointless if someone else rejects the request. The T-junction scenario shown in Figure 5.2a depicts this case. Here either both *B* and *C* should accept *A*'s request, or both should reject it. If only one accepts the request, it will cause a divergence.

Actually, this cannot happen under our communication pattern because of the CONFIRM message. The requesting vehicle will only confirm a global plan that is conflict-free and can enable its requested trajectory. For example, in this T-junction scenario, if one vehicle accepts *A*'s request and the other rejects it, *A* should not be able to confirm. As a result, no one changes its planned trajectory in the end.

Scenario 2 In order to increase the chance to be accepted, one requesting vehicle can send different REQUESTs to different recipients simultaneously. If each accepting vehicle promises to accept a corresponding request, which is unnecessary, and the requesting vehicle confirms all of them, this will cause a divergence. An example is shown in Figure 5.2b where only one vehicle *B* or *C* needs to slow down so that *A* can change its lane. If both *B* and *C* decelerate, it is a waste of time and speed. As we will see in the evaluation, sometimes it can even hinder *A* again.

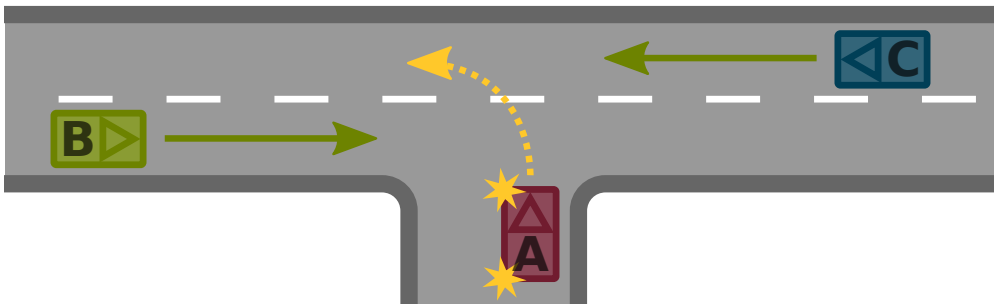
Scenario 3 One accepting vehicle sends different PROMISEs to different requesting vehicles, but in the end, it can satisfy only one of them (although the promised trajectories can look the same). In this case, different requesting vehicles, who may not even be aware of each other, will believe that its own REQUEST is accepted, leading to a conflict in the maneuver.

The latter two cases are similar, and we solve them by requiring that any vehicle should not send different PROMISEs or CONFIRMs within a certain period of time. Multiple different REQUESTs are still allowed. Note that multiple trajectories can appear in one PROMISE, but all of them must be related to the same REQUEST, and the PROMISE must explicitly indicate this REQUEST.

5.4.4 Consequences of Message Losses

The Maneuver Coordination service does not assume a reliable message transmission or message loss detection, because none of these features are provided by the Basic Transport Protocol (BTP) specified by the ETSI in the ITS-G5 standard [24]. So it is possible that during the negotiation, some messages get lost or the communication between vehicles becomes completely broken. In the following, we will analyze the consequences of such communication failures.

Because the CONFIRM is the last message during the negotiation, we only need to focus on the delivery of this message. Otherwise, if the communication problem occurs prior to that, the



(a) Vehicle *A* wants to turn left and requests *B* and *C* to yield their priorities. So they need to reach an agreement, either both slow down, or neither of them does so.

(b) Vehicle *A* requests *C* to slow down so that it can join the lane, assuming that the deceleration of *C* does not cause *B* to decelerate. Alternatively, *A* can also request *B* to slow down. But it is unnecessary to let both *B* and *C* slow down.

Figure 5.2: Two scenarios where an agreement is required.

requesting vehicle should not be able to send the CONFIRM and the negotiation does not have any effect on the maneuver of everyone in the end.

Since the CONFIRM is broadcast by the requesting vehicle to a set of accepting vehicles, there are three possible results:

All accepting vehicles successfully deliver the CONFIRM Then all accepting vehicles will start the maneuvers as in the CONFIRM, and the requesting vehicle can also successfully execute its requested trajectory.

None of the accepting vehicles deliver the CONFIRM Then the requesting vehicle will notice that the accepting vehicles are not changing their maneuvers as they promised. As a result, it cannot execute its requested trajectory, and falls back to the default plan after the timeout.

Only a subset of the accepting vehicles deliver the CONFIRM This case is more complicated. Regarding the accepting vehicles that do not receive the CONFIRM, they do not change their planned maneuvers. Consequently, the requesting vehicle is still unable to execute its requested trajectory, because at least one accepting vehicle is hindering it. As for the accepting vehicles that receive the CONFIRM, they will update their trajectory accordingly, which, however, is meaningless to the requesting vehicle. In this case, some overhead will occur, but it will not lead

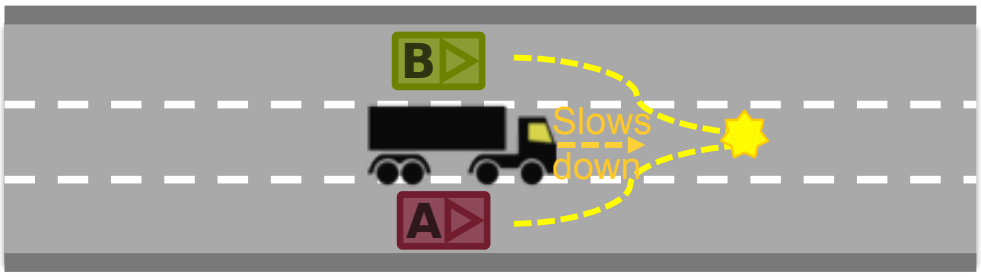


Figure 5.3: The example to explain the necessity of the PROMISE message.

to a catastrophic consequence. This is because of the requirements of the PROMISE and CONFIRM: an accepting vehicle only promises trajectories that is allowed to drive according to the current right-of-way, and all the trajectories in the CONFIRM must be conflict-free.

A special case is that there is only one accepting vehicle involved in the CONFIRM. Then only the first two cases discussed above can happen, leading to the result that either both the requesting vehicle and the accepting vehicle execute the negotiated plan in the CONFIRM, or both abort the negotiation and drive according to the originally planned trajectory. Thus, the impact of communication failures is quite limited.

5.4.5 Discussions and Challenges

In this section, we discuss some considerations of the protocol design. Someone might wonder why the PROMISE and CONFIRM steps are necessary. To illustrate this, we can assume that after receiving the REQUEST, an accepting vehicle immediately changes its maneuver (if it is willing and able to do so) without further communication. This approach has at least two drawbacks.

Firstly, from the viewpoint of the requesting vehicle, it only observes that the accepting vehicle is changing its maneuver, but does not know the reason. It is possible that the accepting vehicle is responding to another vehicle's REQUEST. For example as shown in Figure 5.3, a big truck is driving in the middle lane, and two cars are driving on different sides of it and are not aware of each other. Now both cars request the truck to slow down, so that they can change to the middle lane. If the truck accepts any request of them and decelerates, both cars will believe that they can change to the middle lane, which is conflicting. Therefore, we rely on PROMISE to solve such conflicts. A PROMISE must indicate to which REQUEST it is responding. Consequently, a requesting vehicle knows its REQUEST is accepted only if 1) it receives the PROMISE and 2) the accepting vehicle drives as it promised. In other words, a requesting vehicle must see the accepting vehicle's maneuver change and must know why it is doing so.

Secondly, divergences cannot be prevented if we rely only on REQUESTs, even together with PROMISEs. An accepting vehicle can have several possible trajectories to fulfill a certain REQUEST, and some of them may be conflicting with the promised trajectories of another accepting vehicle. Thus, a consensus mechanism is required to let everyone decide a trajectory that is conflict-free

with each other. This is achieved by the CONFIRM from the requesting vehicle, which acts as a coordinator to resolve conflicts.

As we have seen in the last subsection, a communication failure can cause only a subset of accepting vehicles to change their maneuvers. Another question regarding the protocol is why not adding more communication rounds to establish an "all-to-all" confirmation. However, this is similar to the two general's problem [32] which is unsolvable in a system with unreliable communication links. In the end, there must be such a message, whose sender will immediately take an action without a confirmation while the recipients have to wait for it before taking an action. So adding more rounds to the protocol cannot solve the problem that some accepting vehicles do not receive the last message and thus do not change their maneuvers. Here we choose the requesting vehicle to send this last message and keep the rounds of communication as few as possible.

There are several challenges and open issues that are not covered in this work:

- The requested and promised trajectories only affect the maneuvers after a certain time from the time of sending. The time interval should be enough for the negotiation to be completed. Defining the proper timing relies on additional techniques and knowledge.
- Similarly, the abortion timeout of the requesting vehicle should be properly chosen, such that safety can still be ensured, especially when the requesting vehicle has already started its desired maneuver.
- Each vehicle needs a policy to determine whether it should accept or reject a request. Different metrics can be considered, for example the accumulative speed loss, energy consumption, variation of acceleration, etc. How to define such a policy that is the most beneficial to the overall traffic is another interesting research topic.
- As a preliminary work, we do not allow the maneuver cascading. As introduced previously, cascading means that a vehicle initiates a new maneuver changing request in order to fulfill an existing request from others. If cascading is taken into account, the negotiation will become more complex and the coordination protocol should be adapted accordingly.

These challenges point out the direction for our future work.

5.5 Evaluation

We evaluate the Maneuver Coordination service on the vehicular network simulation framework *Artery*[1]. Artery is based on the *Veins* framework [65] that combines the network simulator

[1] `https://github.com/riebl/artery`

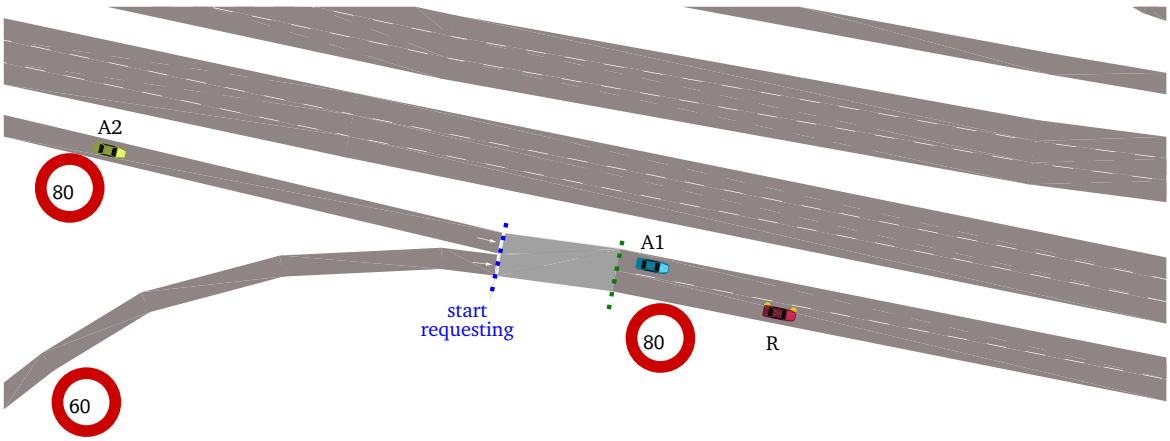


Figure 5.4: The lane-join scenario that models the interchange of Autobahn A2 and A391 (52.3138, 10.52).

OMNeT++ together with the traffic simulator *SUMO*. It also integrates *Vanetza* [59], an open-source implementation of the ETSI ITS-G5 protocol stack. In this section, we describe the implementation of the simulation and the evaluation results.

5.5.1 Implementation

Now we describe the implementation of the simulation from different aspects.

Scenario To demonstrate how the Maneuver Coordination service can help improve the traffic efficiency, we created a lane-join scenario as shown in Figure 5.4, which is modeled after the interchange of the German highway A2 and A391. The road on the bottom-right has a total length of 206 meters and the speed limit is 80 km/h, i .e. 22.2 m/s. The ramp connected to this road on the bottom-left has a speed limit of 60 km/h. The roads are described in an XML file as an input to the simulator.

Vehicles' movement A requesting car R (red) enters the right lane with the speed of 16.67 m/s (60 km/h), and two accepting cars $A1$ (blue) and $A2$ (green) enter the left lane with the full speed of 22.2 m/s. The car R enters the road about 1.2 seconds before $A1$, leading to a 21.5 meters gap between them, and the distance between $A1$ and $A2$ is 84 meters. There is only a single lane by the end of the road, so R must merge into the left lane before it reaches the end.

Communication and Maneuver Coordination service The artery simulator provides a *facility layer* [28] to each vehicle, using vehicular data from SUMO and includes a middleware to support the development of new ITS-G5 services. Therefore, we implement Maneuver Coordination service and register it for all the three vehicles. The messages are sent via ITS-G5's BTP protocol [24].

As mentioned in Section 5.4.2, every vehicle periodically broadcasts negotiation messages together with the planned trajectory. The broadcast interval is set to 100 ms, the same as the lower bound of the CA Service of ITS-G5 [25]. All vehicles are initialized in the detection phase of the Maneuver Coordination service. When a vehicle wants to start a request or it has received certain message(s), its state may change according to the specification of Section 5.4.2. Even though, the vehicle does not send a new message immediately. It must wait until the next broadcasting time. That means, the frequency of message broadcasting of every vehicle is constantly 10 Hz. If a vehicle does not have an update to its state between two broadcasting time points, it keeps broadcasting the same message as last time.

On the lower layer, an IEEE 802.11p radio with an omnidirectional antenna, sending at a base frequency of 5.9 GHz and with a transmission power of 200 mW, is used. The communication channel and range is simulated using the Nakagami Fading model [53].

Simulation process At the beginning, every vehicle is in the detection phase and broadcasts its planned trajectory. Later, vehicle R starts to send REQUEST after it leaves the ramp and enters the straight road (the blue dashed line in Figure 5.4), which is 23.28 seconds since the start of the simulation. This is considered as the start point of the negotiation. The requested trajectories will take effect starting from the position where R can reach after 1 second, i. e. 17.5 m ahead of R's current position (the green dashed line in Figure 5.4). That means, the negotiation must finish in 1 second.

Because at the beginning R has a lower speed than the others, it is unable to directly change lane since the faster $A1$ is blocking it. Now consider the following possibilities:

- Everyone drives in an uncoordinated, selfish manner. That means, the two cars $A1$ and $A2$ just drive with the full speed, and R has to look for an opportunity to join the lane without impacting the others according to the right-of-way rules.
- R requests $A1$ to slow down so that it can overtake $A1$ and change lane in front of it.
- R requests $A2$ to slow down so that it can fit into the middle of $A1$ and $A2$.

Actually, the request is not necessarily realized via the Maneuver Coordination service. For example, the lane-changing model *LC2013* [23] implemented in SUMO already includes the cooperative lane-changing. In LC2013, when a following vehicle sees the turn-signal of the car in front, it can voluntarily adjust its maneuver, e. g. to decelerate, to help the other to change the lane. This implies that the cooperation of LC2013 is a unilateral decision and does not involve

any negotiation mechanism. The willingness of each vehicle to give up its priority to help others is also configurable in the simulator. Thus, LC2013 is a good comparison in the evaluation our Maneuver Coordination service.

It should be pointed out that we focus only on the negotiation mechanism of the Maneuver Coordination service. Thus, the topics such as how the planned, requested and promised trajectories are calculated are not covered in this work. One difficulty is that the Artery simulator does not provide an interface to obtain the planned trajectory from the driver model at runtime. In order to let the Maneuver Coordination service send planned trajectories, we work this around by running the simulation twice. In the first run, the trajectories are recorded while in the second run, the trajectories of the future are broadcasted in the Maneuver Coordination service. Nevertheless, we have to keep in mind that in reality, this should be calculated by each vehicle locally at runtime.

5.5.2 Results of the Speed Changes

In the simulation experiments, we test 6 different cases and depict the speed-time curve of every car in Figure 5.5. The two cases in the first row are the baselines without the Maneuver Coordination service. In the "uncoordinated" case, vehicles are all selfish, while in the latter case, the cooperative lane-join of LC2013 is activated and the willingness to cooperate is set to maximum. As we expected, if there is no cooperation and the right-of-way rules dominate, the requesting car can neither takeover $A1$ nor cut in between $A1$ and $A2$, so it has to almost stop by the end of the road and wait until the other two cars pass by. But surprisingly, the cooperative lane-changing of LC2013 is even worse. $A2$ generously slows down to let R go first, but the latter does not have the confidence to break the right-of-way rule to overtake, so it still stops in the end. This is actually a "misunderstanding" between the two cars due to the lack of communication.

We then test Maneuver Coordination service and let R to make an agreement with one of the two accepting cars (LC2013 cooperative lane-changing is turned off now). It turns out that if $A1$ slows down a little bit, it does not impact $A2$, but can let R quickly overtake. Alternatively, R can ask $A2$ to slow down so that it can cut into the middle of the other two. To do so, R also needs to restrain the speed to adjust its position. The latter seems not as good as the former case, but this can happen when $A1$ does not accept the request because of its own cooperation policy or communication failures.

To stress the importance of the agreement during the coordination, we also simulate two divergence cases where both $A1$ and $A2$ accept the request to slow down. In "divergence1", R believes it has reached the agreement with $A1$. The result is similar to the "coordination1", except that $A2$ also unnecessarily slows down. This divergence seems not very harmful. The bad case occurs in "divergence2", where R wants to cut in between the other two. Originally in "coordination2", only $A2$ slows down to leave a space for R. However, $A1$ now also slows down, making the space not enough for R to merge. As a result, R fails to cut in and has to break in the

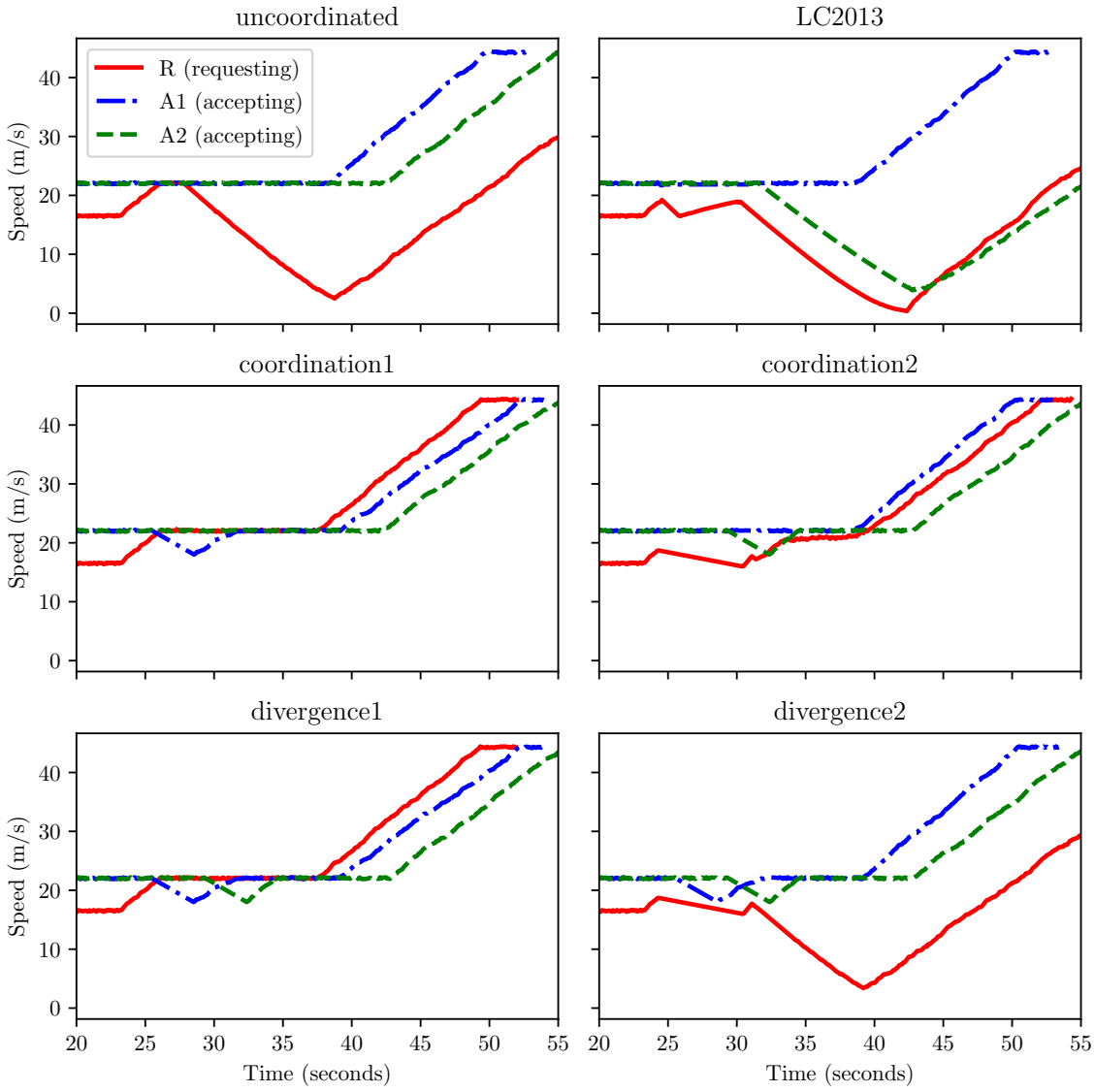


Figure 5.5: The speed of the three vehicles in the lane-join scenario. The "LC2013" refers to the lane-change model adopted by SUMO [23] and with a full willingness of cooperation. In "coordination1", the car R reaches an agreement with $A1$, which accepts to slow down to let R merge in front of it. Similarly in "coordination2", $A2$ accepts R's request and R cuts in between the other two cars. In "divergence1" and "divergence2", both $A1$ and $A2$ accept to slow down. In the former case, R believes it is cooperating with $A1$, while in the latter case, R thinks it is cooperating with $A2$. In fact, the last two divergence cases cannot happen under the coordination protocol. They are only used to show the necessity to avoid divergences.

end. So the divergence not only causes unnecessary speed losses, but can also lead to a complete failure of coordination. Fortunately, the coordination protocol we designed can exclude such divergences by confirming a unique global plan.

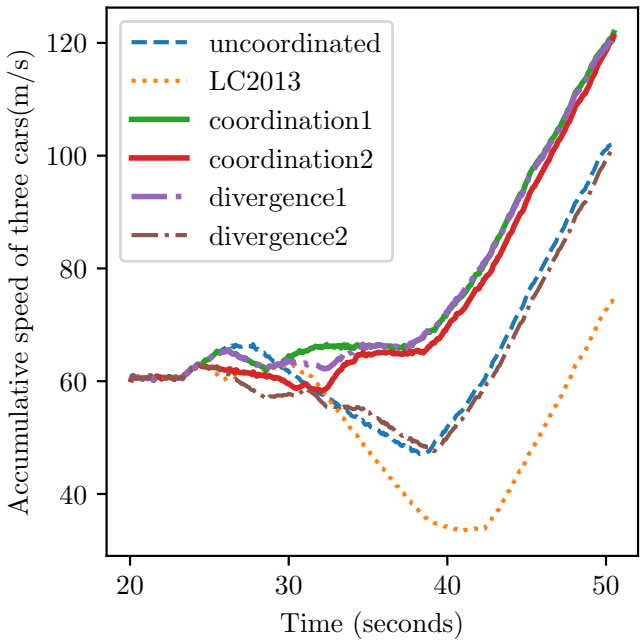


Figure 5.6: The accumulative speed of all the three vehicles.

time loss (s)	requesting car	accepting car 1	accepting car 2	**sum**
unmodified	15.11	2.65	3.03	**20.79**
LC2013	18.53	2.92	14.61	**36.06**
coordination1	3.18	3.78	3.02	**9.98**
coordination2	5.47	2.80	3.45	**11.72**
divergence1	3.17	3.73	3.44	**10.34**
divergence2	15.36	3.26	3.44	**22.06**

Table 5.1: The time loss in seconds.

Figure 5.6 shows the accumulative speed of all three cars. The two successful coordination cases (and the first divergence case) lead to only little speed loss of the impacted accepting vehicles and they recover quickly to the optimal speed. The second divergence is even worse than the non-communicating maneuver. The worst case is the cooperative lane-changing of the LC2013 model. This shows that the effective communication during a coordination is indispensable.

Table 5.1 summarizes the time loss of all cases gathered by SUMO. This metric indicates the waste of the time caused by driving below the ideal speed. As we can see, the Maneuver Coordination service can reduce the total time loss to around 50% compared to the non-communicating driving model.

5.5.3 Communication Latency

In the simulation, if no package loss happens, the vehicle R eventually chooses the promise of $A1$, and it takes 360 ms until $A1$ receives the final confirm of R. This result corresponds to our expectation, as it is slightly more than 3 broadcasting intervals ($3 * 100$ ms). We also injected a package loss rate of 30%, and this time it takes 760 ms on average until someone receives the confirm. Because it is less than 1 second, the chosen accepting vehicle can still cooperate with R successfully.

5.6 Conclusion

We propose the coordination protocol as an extension to the Maneuver Coordination service [42] for autonomous vehicles to coordinate their driving behaviors more effectively. The coordination enables vehicles to cooperate in a spontaneous group and work out better driving plans which can improve the traffic efficiency. On one side, the protocol allows each vehicle to propose multiple trajectories simultaneously at the beginning of the coordination, which can increase the probability to find a feasible joint plan. On the other side, it can guarantee an agreement and avoid divergences where different vehicles decide contradictory trajectories in the end. We analyze the impact of potential communication failures and show that in some cases the impact is quite small, while in other cases some overhead can occur but the safety can still be ensured. We tested our approach using the Artery simulation framework with a lane-join scenario. The result shows that the Maneuver Coordination service can save up to 50% time loss compared to the case where no coordination exists.

6 Summary

This thesis investigates the fault-tolerant consensus problem on wireless embedded systems. Although consensus is a classical topic in distributed system, it turns out that there are new considerations and challenges for wireless embedded systems. Among different fault models, we choose the hybrid fault model thanks to the development of trusted computing technology and dedicated hardware. This chapter will summarize our findings and suggest the direction of future works.

6.1 Research Questions and Answers

In this section, we will summarize the research questions and corresponding contributions of this thesis.

Hybrid fault-tolerant consensus, especially by utilizing a trusted subsystem, is studied in several previous works, but none of them has considered randomized consensus in a fully synchronous system. Thus, the first research question is:

- **Is there a randomized, asynchronous consensus algorithm that can tolerate $\lfloor \frac{n-1}{2} \rfloor$ arbitrarily faulty processes by utilizing a trusted subsystem?**

We have designed TRUSTED BEN-OR, a binary consensus algorithm that can tolerate up to $\lfloor \frac{n-1}{2} \rfloor$ faulty processes, which is the best bound in an asynchronous system. It utilizes a trusted monotonic counter in each process to prevent equivocation attacks, and a trusted random number generator as a reliable source of randomness. TRUSTED BEN-OR can work in a fully asynchronous system. We have elaborately designed the mechanism of message certificate, and proved its correctness even against a strong adversary. We also discussed some common flaws when we design a

randomized consensus algorithm and prove its termination, especially when a strong adversary exists. TRUSTED BEN-OR is designed for wireless embedded systems. It uses multicast to make full use of the communication medium, and does not rely on reliable communication channels such as TCP. It can also tolerate limited message omission failures. We evaluate the performance of TRUSTED BEN-OR in a testbed of 10 RaspberryPis and injected Byzantine faults as well as up to 60% message omission faults, then compare it with another Byzantine fault-tolerant consensus for wireless embedded systems. The results show that TRUSTED BEN-OR outperforms in almost all cases, while tolerating more faulty processes ($\lfloor \frac{n-1}{2} \rfloor$ compared to $\lfloor \frac{n-1}{3} \rfloor$) with the same group size n.

Furthermore, most of the previous hybrid fault-tolerant algorithms are designed as state machine replication protocols for servers. They can hardly be applied on embedded systems. Some of them also have issues such as the unlimited memory usage, as discussed in Chapter 4.2.2. The next research question is:

- **Is there a multi-value hybrid fault-tolerant consensus algorithm that is tailored for wireless embedded systems?**

RATCHETA is designed to answer this question. It is a multi-value consensus algorithm that works in partially synchronous system and can also tolerate $\lfloor \frac{n-1}{2} \rfloor$ faulty processes. We use a pair of trusted counters to prevent equivocation. Compared with other algorithms that use only one counter, using two counters can significantly simplify the algorithm design and proof, and can overcome the unlimited memory usage issue. Similar to TRUSTED BEN-OR, the RATCHETA algorithm is also tailored for wireless embedded system. It uses multicast as well and does not rely on reliable communication either. The algorithm is tested on the same testbed and it can effectively tolerate Byzantine faults and limited omission faults.

Since consensus is a fundamental problem of distributed systems, many applications can be built on top of a consensus service. Naturally, we want to ask:

- **How can fault-tolerant consensus be used in applications of wireless embedded systems?**

Autonomous vehicle together with vehicular network is a cutting edge technology. Accordingly, we have designed an algorithm of Maneuver Coordination service that leverages V2V communication to coordinate driving trajectories among autonomous vehicles. The algorithm can be abstracted as a one-round consensus and can avoid potential disagreement among the communicating vehicles. As a result, safety can be guaranteed. We have conducted experiments on a simulation framework. The results show that the Maneuver Coordination service can increase traffic efficiency by reducing time loss caused by uncoordinated driving.

6.2 Outlook

This thesis focuses on the design and proof of the hybrid fault-tolerant consensus algorithms, but we cannot cover all aspects of this research topic. We suggest that the future works can focus on **building more applications based on hybrid fault-tolerant consensus**. TRUSTED BEN-OR and RATCHETA are general purposed consensus algorithms and can be used as middleware to build more complex applications for wireless embedded systems. The consensus for Maneuver Coordination service is such an example. Further use cases can include:

- Sensor fusion: inputs from different sensors are merged as a global knowledge. Here median validity can also be applied.
- Vehicle and UAV platooning: a group of vehicles or UAVs move together in a coordinated manner. Unlike the Maneuver Coordination service which happens instantly in a spontaneous group, platooning is normally long-lasting and requires continuous consensus.
- Robotics: several robots work cooperatively with a common goal, for example the life-searching mission as we discussed in Chapter 4.2.

Here we just name a few examples. More use case scenarios can be explored.

References

[1] ETSI EN 102 636-5-1. *Intelligent transport systems (ITS); Vehicular Communications; GeoNetworking; Part 5: Transport Protocols; Sub-part 1: Basic Transport Protocol*. 2017.

[2] Ittai Abraham, Danny Dolev, and Joseph Y Halpern. "An almost-surely terminating polynomial protocol for asynchronous byzantine agreement with optimal resilience". In: *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*. ACM. 2008, pp. 405–414.

[3] Marcos K Aguilera and Sam Toueg. "The correctness proof of Ben-Or's randomized consensus algorithm". In: *Distributed Computing* 25.5 (2012), pp. 371–381.

[4] Buke Ao, Yongcai Wang, Lu Yu, Richard R Brooks, and SS Iyengar. "On precision bound of distributed fault-tolerant sensor fusion algorithms". In: *ACM Computing Surveys (CSUR)* 49.1 (2016), p. 5.

[5] ARM. *ARM Security Technology - Building a Secure System using TrustZone Technology*. Tech. rep. PRD29-GENC-009492C. ARM Technical White Paper, 2009.

[6] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. "Hybrids on Steroids: SGX-Based High Performance BFT". In: *Proceedings of the Twelfth European Conference on Computer Systems*. ACM. 2017, pp. 222–237.

[7] Michael Ben-Or. "Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols". In: *Proceedings of the second annual ACM symposium on Principles of distributed computing*. ACM. 1983, pp. 27–30.

[8] boost.org. *Boost.Asio*. https://www.boost.org/doc/libs/1_65_0/$doc/html/boost_asio.html$. visited in Oktober, 2020.

[9] Gabriel Bracha. "An asynchronous [(n-1)/3]-resilient consensus protocol". In: *Proceedings of the third annual ACM symposium on Principles of distributed computing*. ACM. 1984, pp. 154–162.

[10] Gabriel Bracha and Sam Toueg. "Resilient consensus protocols". In: *Proceedings of the second annual ACM symposium on Principles of distributed computing*. ACM. 1983, pp. 12–26.

[11] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. "Rollback and forking detection for trusted execution environments using lightweight collective memory". In: *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2017, pp. 157–168.

[12] Alberto Broggi, Pietro Cerri, Mirko Felisa, Maria Chiara Laghi, Luca Mazzei, and Pier Paolo Porta. "The VisLab Intercontinental Autonomous Challenge: an extensive test for a platoon of intelligent vehicles". In: *International Journal of Vehicle Autonomous Systems* 10.3 (2012), pp. 147–164.

[13] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011, pp. 4–7.

[14] Ran Canetti and Tal Rabin. "Fast asynchronous Byzantine agreement with optimal resilience". In: *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. ACM. 1993, pp. 42–51.

[15] Miguel Castro and Barbara Liskov. "Practical Byzantine fault tolerance and proactive recovery". In: *ACM Transactions on Computer Systems* 20.4 (2002), pp. 398–461.

[16] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. "Attested append-only memory: Making adversaries stick to their word". In: *ACM SIGOPS Operating Systems Review*. Vol. 41. 6. ACM. 2007, pp. 189–204.

[17] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. "From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures". In: *The Computer Journal* 49.1 (2006), pp. 82–96.

[18] Miguel Correia, Giuliana S Veronese, and Lau Cheuk Lung. "Asynchronous Byzantine consensus with 2f+ 1 processes". In: *Proceedings of the 2010 ACM symposium on applied computing*. ACM. 2010, pp. 475–480.

[19] J. Q. Cui, S. K. Phang, K. Z. Y. Ang, F. Wang, X. Dong, Y. Ke, S. Lai, K. Li, X. Li, F. Lin, J. Lin, P. Liu, T. Pang, B. Wang, K. Wang, Z. Yang, and B. M. Chen. "Drones for cooperative search and rescue in post-disaster situation". In: *2015 IEEE 7th International Conference on Cybernetics and Intelligent Systems (CIS) and IEEE Conference on Robotics, Automation and Mechatronics (RAM)*. 2015, pp. 167–174. DOI: 10.1109/ICCIS.2015.7274615.

[20] Shi-Lu Dai, Shude He, Hai Lin, and Cong Wang. "Platoon formation control with prescribed performance guarantees for USVs". In: *IEEE Transactions on Industrial Electronics* 65.5 (2018), pp. 4237–4246.

[21] Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl. "Reaching Approximate Agreement in the Presence of Faults". In: *J. ACM* 33.3 (May 1986), pp. 499–516. ISSN: 0004-5411. DOI: 10.1145/5925.5931.

[22] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. “Consensus in the presence of partial synchrony”. In: *Journal of the ACM (JACM)* 35.2 (1988), pp. 288–323.

[23] Jakob Erdmann. “SUMO’s lane-changing model”. In: *Modeling Mobility with Open Data*. Springer, 2015, pp. 105–123.

[24] *ETSI EN 302 636-5-1 V1.2.0 - Intelligent Transport Systems (ITS); Vehicular Communications; GeoNetworking; Part 5: Transport Protocols; Sub-part 1: Basic Transport Protocol*. ETSI, Aug. 2014.

[25] *ETSI EN 302 637-2 V1.3.1 - Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Part 2: Specification of Cooperative Awareness Basic Service*. ETSI, Sept. 2014.

[26] *ETSI EN 302 665 V1.1.1 - Intelligent Transport Systems; Communications Architecture*. ETSI, Sept. 2010.

[27] *ETSI TR 103 562 V2.1.1 - Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Analysis of the Collective Perception Service*. ETSI, Dec. 2019.

[28] *ETSI TS 102 894-1 V1.1.1 - Intelligent Transport Systems (ITS); Users and applications requirements; Part 1: Facility layer structure, functional requirements and specifications*. ETSI, Aug. 2013.

[29] *ETSI TS 103 324 V0.0.19 - Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Specification of the Collective Perception Service*. ETSI, July 2020.

[30] *ETSI TS 302 890-2 V0.0.3 - Intelligent Transport Systems (ITS); Facilities Layer Function; Part 2: Position and Time Facility Specification*. ETSI, Mar. 2019.

[31] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. “Impossibility of distributed consensus with one faulty process”. In: *Journal of the ACM (JACM)* 32.2 (1985), pp. 374–382.

[32] J. N. Gray. “Notes on data base operating systems”. In: *Operating Systems: An Advanced Course*. Ed. by R. Bayer, R. M. Graham, and G. Seegmüller. Berlin, Heidelberg: Springer Berlin Heidelberg, 1978, pp. 393–481. ISBN: 978-3-540-35880-0. DOI: 10.1007/3-540-08755-9_9.

[33] Hendrik-Jörn Günther, Björn Mennenga, Oliver Trauer, Raphael Riebl, and Lars Wolf. “Realizing collective perception in a vehicle”. In: *2016 IEEE Vehicular Networking Conference (VNC)*. IEEE. 2016, pp. 1–8.

[34] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. “Flexible paxos: Quorum intersection revisited”. In: *arXiv preprint arXiv:1608.06696* (2016).

[35] Zafar Iqbal, Kiseon Kim, and Heung-No Lee. “A cooperative wireless sensor network for indoor industrial monitoring”. In: *IEEE Transactions on Industrial Informatics* 13.2 (2017), pp. 482–491.

[36] *ISO/TS 19091:2019 - Intelligent transport systems – Cooperative ITS - Using V2I and I2V communications for applications related to signalized intersections*. ISO, 2019.

[37] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. “CheapBFT: Resource-efficient Byzantine Fault Tolerance”. In: *Proceedings of the EuroSys 2012 Conference*. Ed. by European Chapter of ACM SIGOPS. Switzerland, 2012, pp. 295–308. ISBN: 978-1-4503-1223-3.

[38] Elham Semsar Kazerooni and Jeroen Ploeg. “Interaction protocols for cooperative merging and lane reduction scenarios”. In: *2015 IEEE 18th International Conference on Intelligent Transportation Systems*. IEEE, 2015, pp. 1964–1970.

[39] Leslie Lamport et al. “Paxos made simple”. In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[40] Leslie Lamport, Robert Shostak, and Marshall Pease. “The Byzantine Generals Problem”. In: *ACM Trans. Program. Lang. Syst.* 4.3 (July 1982), pp. 382–401. ISSN: 0164-0925. DOI: 10.1145/357172.357176.

[41] Heath J LeBlanc, Haotian Zhang, Xenofon Koutsoukos, and Shreyas Sundaram. “Resilient asymptotic consensus in robust networks”. In: *IEEE Journal on Selected Areas in Communications* 31.4 (2013), pp. 766–781.

[42] Bernd Lehmann, Hendrik-Jörn Günther, and Lars Wolf. “A Generic Approach towards Maneuver Coordination for Automated Vehicles”. In: *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*. IEEE. 2018, pp. 3333–3339.

[43] Dave Levin, John R Douceur, Jacob R Lorch, and Thomas Moscibroda. “TrInc: Small Trusted Hardware for Large Distributed Systems.” In: *NSDI*. Vol. 9. 2009, pp. 1–14.

[44] Lin Li, Ruochen Hao, Wanjing Ma, Xinzhou Qi, and Chenxue Diao. *Swarm Intelligence Based Algorithm for Management of Autonomous Vehicles on Arterials*. Tech. rep. SAE Technical Paper, 2018.

[45] Kuo-Yun Liang, Jonas Mårtensson, and Karl H Johansson. “Heavy-duty vehicle platoon formation for fuel efficiency”. In: *IEEE Transactions on Intelligent Transportation Systems* 17.4 (2016), pp. 1051–1061.

[46] Todd Litman. *Autonomous vehicle implementation predictions*. Victoria Transport Policy Institute Victoria, Canada, 2017.

[47] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. “XFT: Practical Fault Tolerance beyond Crashes.” In: *OSDI*. 2016, pp. 485–500.

[48] Greg Marsden, Mike McDonald, and Mark Brackstone. “Towards an understanding of adaptive cruise control”. In: *Transportation Research Part C: Emerging Technologies* 9.1 (2001), pp. 33–51.

[49] Friedemann Mattern and Christian Floerkemeier. “From the Internet of Computers to the Internet of Things”. In: *From active data management to event-based systems and more*. Springer, 2010, pp. 242–259.

[50] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. “The honey badger of BFT protocols”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 31–42.

[51] Henrique Moniz, Nuno Ferreira Neves, and Miguel Correia. “Turquois: Byzantine consensus in wireless ad hoc networks”. In: *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*. IEEE. 2010, pp. 537–546.

[52] Henrique Moniz, Nuno Ferreira Neves, Miguel Correia, and Paulo Veríssimo. “Randomization can be a healer: Consensus with dynamic omission failures”. In: *International Symposium on Distributed Computing*. Springer. 2009, pp. 63–77.

[53] Minoru Nakagami. “The m-distribution — A general formula of intensity distribution of rapid fading”. In: *Statistical methods in radio wave propagation*. Elsevier, 1960, pp. 3–36.

[54] Gil Neiger. “Distributed consensus revisited”. In: *Information Processing Letters* 49.4 (1994), pp. 195–201. ISSN: 0020-0190. DOI: http://dx.doi.org/10.1016/0020-0190(94)90011-6.

[55] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. “Trustzone explained: Architectural features and use cases”. In: *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*. IEEE. 2016, pp. 445–451.

[56] Reza Olfati-Saber, J Alex Fax, and Richard M Murray. “Consensus and cooperation in networked multi-agent systems”. In: *Proceedings of the IEEE* 95.1 (2007), pp. 215–233.

[57] ESA Earth Online. *Swarm*. https://earth.esa.int/web/guest/missions/esa-operational-eo-missions/swarm. visited in April, 2018.

[58] Marshall Pease, Robert Shostak, and Leslie Lamport. “Reaching agreement in the presence of faults”. In: *Journal of the ACM (JACM)* 27.2 (1980), pp. 228–234.

[59] Raphael Riebl, Hendrik-Jörn Günther, Christian Facchi, and Lars Wolf. “Artery: Extending veins for vanet applications”. In: *Models and Technologies for Intelligent Transportation Systems (MT-ITS), 2015 International Conference on*. IEEE, 2015, pp. 450–456.

[60] Stefania Santini, Alessandro Salvi, Antonio Saverio Valente, Antonio Pescapè, Michele Segata, and R Lo Cigno. “A consensus-based approach for platooning with inter-vehicular communications”. In: *Computer Communications (INFOCOM), 2015 IEEE Conference on*. IEEE. 2015, pp. 1158–1166.

[61] Nicola Santoro and Peter Widmayer. “Time is not a healer”. In: *STACS 89* (1989), pp. 304–313.

[62] Luca Schenato and Federico Fiorentin. “Average TimeSynch: A consensus-based protocol for clock synchronization in wireless sensor networks”. In: *Automatica* 47.9 (2011), pp. 1878–1886.

[63] Fred B Schneider. “Implementing fault-tolerant services using the state machine approach: A tutorial”. In: *ACM Computing Surveys (CSUR)* 22.4 (1990), pp. 299–319.

[64] Steven E Shladover. “Automated vehicles for highway operations (automated highway systems)”. In: *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering* 219.1 (2005), pp. 53–75.

[65] C. Sommer, R. German, and F. Dressler. “Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis”. In: *IEEE Trans. Mobile Comput.* 10.1 (Jan. 2011), pp. 3–15. DOI: 10.1109/TMC.2010.133.

[66] David Stolz and Roger Wattenhofer. “Byzantine Agreement with Median Validity”. In: *19th International Conference on Priniciples of Distributed Systems (OPODIS), Rennes, France*. 2015.

[67] TrustedFirmware.org. *OP-TEE Documentation*. https://optee.readthedocs.io. visited in October, 2020.

[68] Bart Van Arem, Cornelie JG Van Driel, and Ruben Visser. “The impact of cooperative adaptive cruise control on traffic-flow characteristics”. In: *IEEE Transactions on intelligent transportation systems* 7.4 (2006), pp. 429–436.

[69] Bruno Vavala and Nuno Neves. “Robust and speculative Byzantine randomized consensus with constant time complexity in normal conditions”. In: *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*. IEEE. 2012, pp. 161–170.

[70] Paulo E Veríssimo. “Travelling through wormholes: a new look at distributed systems models”. In: *ACM SIGACT News* 37.1 (2006), pp. 66–81.

[71] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Veríssimo. “Efficient byzantine fault-tolerance”. In: *Computers, IEEE Transactions on* 62.1 (2013), pp. 16–30.

[72] Vladimir Vukadinovic, Krzysztof Bakowski, Patrick Marsch, Ian Dexter Garcia, Hua Xu, Michal Sybis, Pawel Sroka, Krzysztof Wesolowski, David Lister, and Ilaria Thibault. “3GPP C-V2X and IEEE 802.11 p for Vehicle-to-Vehicle communications in highway platooning scenarios”. In: *Ad Hoc Networks* 74 (2018), pp. 17–29.

[73] Le Yi Wang, Ali Syed, Gang George Yin, Abhilash Pandya, and Hongwei Zhang. “Control of vehicle platoons for highway safety and efficient utility: Consensus with communications and vehicle dynamics”. In: *Journal of Systems Science and Complexity* 27.4 (2014), pp. 605–631. ISSN: 1559-7067. DOI: 10.1007/s11424-014-2115-z.

[74] Ziran Wang, Guoyuan Wu, and Matthew Barth. *Distributed consensus-based cooperative highway on-ramp merging using V2X communications*. Tech. rep. SAE Technical Paper, 2018.

[75] Guanghui Wen, Zhisheng Duan, Guanrong Chen, and Wenwu Yu. “Consensus tracking of multi-agent systems with Lipschitz-type node dynamics and switching topologies”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 61.2 (2014), pp. 499–511.

[76] Moritz Werling, Julius Ziegler, Sören Kammel, and Sebastian Thrun. “Optimal trajectory generation for dynamic street scenarios in a frenet frame”. In: *2010 IEEE International Conference on Robotics and Automation*. IEEE. 2010, pp. 987–993.

[77] Lin Xiao, Stephen Boyd, and Sanjay Lall. “A scheme for robust distributed sensor fusion based on average consensus”. In: *Proceedings of the 4th international symposium on Information processing in sensor networks*. IEEE Press. 2005, p. 9.

[78] W. Xu and R. Kapitza. “RATCHETA: Memory-Bounded Hybrid Byzantine Consensus for Cooperative Embedded Systems”. In: *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*. Oct. 2018, pp. 103–112. DOI: 10.1109/SRDS.2018.00021.

[79] W. Xu, M. Wegner, L. Wolf, and R. Kapitza. “Byzantine Agreement Service for Cooperative Wireless Embedded Systems”. In: *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. June 2017, pp. 10–15. DOI: 10.1109/DSN-W.2017.45.

[80] W. Xu, A. Willecke, M. Wegner, L. Wolf, and R. Kapitza. “Autonomous Maneuver Coordination Via Vehicular Communication”. In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. June 2019, pp. 70–77. DOI: 10.1109/DSN-W.2019.00022.

[81] Wenbo Xu, Signe Rüsch, Bijun Li, and Rüdiger Kapitza. “Hybrid Fault-Tolerant Consensus in Asynchronous and Wireless Embedded Systems”. In: *22nd International Conference on Principles of Distributed Systems (OPODIS 2018)*. Ed. by Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira. Vol. 125. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 15:1–15:16. ISBN: 978-3-95977-098-9. DOI: 10.4230/LIPIcs.OPODIS.2018.15.